Arm® A64 Instruction Set Architecture, for Arm A-class
Future Architecture Technologies in the A architecture profile
Product Status

This release covers multiple versions of the architecture. The content relating to different versions is given different quality ratings.

The information related to v8.7 of the architecture and features FEAT_CSRE and FEAT_BRBE are at Alpha quality. Alpha quality means that most major features of the specification are described in the manual, some features and details might be missing.

The information related to versions of the architecture introduced before v8.7 of the architecture and for the remaining features (FEAT_SVE2, FEAT_TME, FEAT_TRBE and FEAT_ETE), which was also published in previous releases, is at BET quality. BET quality means that all major features of the specification are described, some details might be missing.

Web Address

http://www.arm.com
A64 -- Base Instructions (alphabetic order)

**ADC**: Add with Carry.

**ADCS**: Add with Carry, setting flags.

**ADD (extended register)**: Add (extended register).

**ADD (immediate)**: Add (immediate).

**ADD (shifted register)**: Add (shifted register).

**ADDG**: Add with Tag.

**ADDS (extended register)**: Add (extended register), setting flags.

**ADDS (immediate)**: Add (immediate), setting flags.

**ADDS (shifted register)**: Add (shifted register), setting flags.

**ADR**: Form PC-relative address.

**ADRP**: Form PC-relative address to 4KB page.

**AND (immediate)**: Bitwise AND (immediate).

**AND (shifted register)**: Bitwise AND (shifted register).

**ANDS (immediate)**: Bitwise AND (immediate), setting flags.

**ANDS (shifted register)**: Bitwise AND (shifted register), setting flags.

**ASR (immediate)**: Arithmetic Shift Right (immediate): an alias of SBFM.

**ASR (register)**: Arithmetic Shift Right (register): an alias of ASRV.

**ASRV**: Arithmetic Shift Right Variable.

**AT**: Address Translate: an alias of SYS.

**AUTDA, AUTDZA**: Authenticate Data address, using key A.

**AUTDB, AUTDZB**: Authenticate Data address, using key B.

**AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA**: Authenticate Instruction address, using key A.

**AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB**: Authenticate Instruction address, using key B.

**AXFLAG**: Convert floating-point condition flags from Arm to external format.

**B**: Branch.

**B.cond**: Branch conditionally.

**BFC**: Bitfield Clear: an alias of BFM.

**BFI**: Bitfield Insert: an alias of BFM.

**BFM**: Bitfield Move.

**BFXIL**: Bitfield extract and insert at low end: an alias of BFM.

**BIC (shifted register)**: Bitwise Bit Clear (shifted register).

**BICS (shifted register)**: Bitwise Bit Clear (shifted register), setting flags.

**BL**: Branch with Link.

**BLR**: Branch with Link to Register.
BLRAA, BLRAAZ, BLRAB, BLRABZ: Branch with Link to Register, with pointer authentication.

BR: Branch to Register.

BRAA, BRAAZ, BRAB, BRABZ: Branch to Register, with pointer authentication.

BRK: Breakpoint instruction.

BTI: Branch Target Identification.

CAS, CASA, CASAL, CASL: Compare and Swap word or doubleword in memory.

CASB, CASAB, CASALB, CASLB: Compare and Swap byte in memory.

CASH, CASAH, CASALH, CASLH: Compare and Swap halfword in memory.

CASP, CASPA, CASPAL, CASPL: Compare and Swap Pair of words or doublewords in memory.

CBNZ: Compare and Branch on Nonzero.

CBZ: Compare and Branch on Zero.

CCMN (immediate): Conditional Compare Negative (immediate).

CCMN (register): Conditional Compare Negative (register).

CCMP (immediate): Conditional Compare (immediate).

CCMP (register): Conditional Compare (register).

CFINV: Invert Carry Flag.

CFP: Control Flow Prediction Restriction by Context: an alias of SYS.

CINC: Conditional Increment: an alias of CSINC.

CINV: Conditional Invert: an alias of CSINV.

CLREX: Clear Exclusive.

CLS: Count Leading Sign bits.

CLZ: Count Leading Zeros.


CMN (immediate): Compare Negative (immediate): an alias of ADDS (immediate).


CMP (immediate): Compare (immediate): an alias of SUBS (immediate).


CMPP: Compare with Tag: an alias of SUBPS.

CNEG: Conditional Negate: an alias of CSNEG.

CPP: Cache Prefetch Prediction Restriction by Context: an alias of SYS.


CRC32CB, CRC32CH, CRC32CW, CRC32CX: CRC32C checksum.

CSDB: Consumption of Speculative Data Barrier.

CSEL: Conditional Select.

CSET: Conditional Set: an alias of CSINC.
CSETM: Conditional Set Mask: an alias of CSINV.

CSINC: Conditional Select Increment.

CSINV: Conditional Select Invert.

CSNEG: Conditional Select Negation.

DC: Data Cache operation: an alias of SYS.

DCPS1: Debug Change PE State to EL1..

DCPS2: Debug Change PE State to EL2..

DCPS3: Debug Change PE State to EL3.

DGH: Data Gathering Hint.

DMB: Data Memory Barrier.

DRPS: Debug restore process state.

DSB: Data Synchronization Barrier.

DVP: Data Value Prediction Restriction by Context: an alias of SYS.

EON (shifted register): Bitwise Exclusive OR NOT (shifted register).

EOR (immediate): Bitwise Exclusive OR (immediate).

EOR (shifted register): Bitwise Exclusive OR (shifted register).

ERET: Exception Return.

ERETAA, ERETAB: Exception Return, with pointer authentication.

ESB: Error Synchronization Barrier.

EXTR: Extract register.

GMI: Tag Mask Insert.

HINT: Hint instruction.

HLT: Halt instruction.

HVC: Hypervisor Call.

IC: Instruction Cache operation: an alias of SYS.

IRG: Insert Random Tag.

ISB: Instruction Synchronization Barrier.

LD64B: Single-copy Atomic 64-byte Load.

LDADD, LDADDA, LDADDAL, LDADDL: Atomic add on word or doubleword in memory.

LDADDB, LDADDAB, LDADDALB, LDADDLB: Atomic add on byte in memory.

LDADDH, LDADDAH, LDADDALH, LDADDLH: Atomic add on halfword in memory.

LDAPR: Load-Acquire RCpc Register.

LDAPRB: Load-Acquire RCpc Register Byte.

LDAPRH: Load-Acquire RCpc Register Halfword.

LDAPUR: Load-Acquire RCpc Register (unscaled).

LDAPURB: Load-Acquire RCpc Register Byte (unscaled).
LDAPURH: Load-Acquire RCpc Register Halfword (unscaled).
LDAPURSB: Load-Acquire RCpc Register Signed Byte (unscaled).
LDAPURSH: Load-Acquire RCpc Register Signed Halfword (unscaled).
LDAPURSW: Load-Acquire RCpc Register Signed Word (unscaled).
LDAR: Load-Acquire Register.
LDARB: Load-Acquire Register Byte.
LDARH: Load-Acquire Register Halfword.
LDAXP: Load-Acquire Exclusive Pair of Registers.
LDAXR: Load-Acquire Exclusive Register.
LDAXRB: Load-Acquire Exclusive Register Byte.
LDAXRH: Load-Acquire Exclusive Register Halfword.
LDCLR, LDCLRA, LDCLRAL, LDCLRL: Atomic bit clear on word or doubleword in memory.
LDCLR, LDCLRAB, LDCLRALB, LDCLRLB: Atomic bit clear on byte in memory.
LDCLRH, LDCLRALH, LDCLRLH: Atomic bit clear on halfword in memory.
LDEOR, LDEORA, LDEORAL, LDEORL: Atomic exclusive OR on word or doubleword in memory.
LDEORB, LDEORAB, LDEORALB, LDEORLB: Atomic exclusive OR on byte in memory.
LDEORH, LDEORAH, LDEORALH, LDEORLH: Atomic exclusive OR on halfword in memory.
LDG: Load Allocation Tag.
LDGM: Load Tag Multiple.
LDLAR: Load LOAcquire Register.
LDLARB: Load LOAcquire Register Byte.
LDLARH: Load LOAcquire Register Halfword.
LDNP: Load Pair of Registers, with non-temporal hint.
LDP: Load Pair of Registers.
LDPSW: Load Pair of Registers Signed Word.
LDR (immediate): Load Register (immediate).
LDR (literal): Load Register (literal).
LDR (register): Load Register (register).
LDRAA, LDRAB: Load Register, with pointer authentication.
LDRB (immediate): Load Register Byte (immediate).
LDRB (register): Load Register Byte (register).
LDRH (immediate): Load Register Halfword (immediate).
LDRH (register): Load Register Halfword (register).
LDRSB (immediate): Load Register Signed Byte (immediate).
LDRSB (register): Load Register Signed Byte (register).
LDRSH (immediate): Load Register Signed Halfword (immediate).
LDRSH (register): Load Register Signed Halfword (register).

LDRSW (immediate): Load Register Signed Word (immediate).

LDRSW (literal): Load Register Signed Word (literal).

LDRSW (register): Load Register Signed Word (register).

LDSET, LDSETA, LDSETL, LDSETL: Atomic bit set on word or doubleword in memory.

LDSETB, LDSETAB, LDSETLB, LDSETLB: Atomic bit set on byte in memory.

LDSETH, LDSETHA, LDSETLH, LDSETHL: Atomic bit set on halfword in memory.

LDMAX, LDMAXA, LDMAXAL, LDMAXL: Atomic signed maximum on word or doubleword in memory.

LDMAXB, LDMAXAB, LDMAXALB, LDMAXLB: Atomic signed maximum on byte in memory.

LDMAXH, LDMAXAH, LDMAXALH, LDMAXLH: Atomic signed maximum on halfword in memory.

LDMIN, LDMINA, LDMINAL, LDMINL: Atomic signed minimum on word or doubleword in memory.

LDMINB, LDMINAB, LDMINALB, LDMINLB: Atomic signed minimum on byte in memory.

LDMINH, LDMINAH, LDMINALH, LDMINLH: Atomic signed minimum on halfword in memory.

LDTR: Load Register (unprivileged).

LDTRB: Load Register Byte (unprivileged).

LDTRH: Load Register Halfword (unprivileged).

LDTRSB: Load Register Signed Byte (unprivileged).

LDTRSH: Load Register Signed Halfword (unprivileged).

LDTRSW: Load Register Signed Word (unprivileged).

LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL: Atomic unsigned maximum on word or doubleword in memory.

LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB: Atomic unsigned maximum on byte in memory.

LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH: Atomic unsigned maximum on halfword in memory.

LDUMIN, LDUMINA, LDUMINAL, LDUMINL: Atomic unsigned minimum on word or doubleword in memory.

LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB: Atomic unsigned minimum on byte in memory.

LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH: Atomic unsigned minimum on halfword in memory.

LDUR: Load Register (unscaled).

LDURB: Load Register Byte (unscaled).

LDURH: Load Register Halfword (unscaled).

LDURSB: Load Register Signed Byte (unscaled).

LDURSH: Load Register Signed Halfword (unscaled).

LDURSW: Load Register Signed Word (unscaled).

LDXP: Load Exclusive Pair of Registers.

LDXR: Load Exclusive Register.

LDXRB: Load Exclusive Register Byte.

LDXRH: Load Exclusive Register Halfword.

LSL (immediate): Logical Shift Left (immediate): an alias of UBFM.
**LSL (register)**: Logical Shift Left (register): an alias of LSLV.

**LSLV**: Logical Shift Left Variable.

**LSR (immediate)**: Logical Shift Right (immediate): an alias of UBFM.

**LSR (register)**: Logical Shift Right (register): an alias of LSRV.

**LSRV**: Logical Shift Right Variable.

**MADD**: Multiply-Add.

**MNEG**: Multiply-Negate: an alias of MSUB.

**MOV (bitmask immediate)**: Move (bitmask immediate): an alias of ORR (immediate).

**MOV (inverted wide immediate)**: Move (inverted wide immediate): an alias of MOVN.

**MOV (register)**: Move (register): an alias of ORR (shifted register).

**MOV (to/from SP)**: Move between register and stack pointer: an alias of ADD (immediate).

**MOV (wide immediate)**: Move (wide immediate): an alias of MOVZ.

**MOVK**: Move wide with keep.

**MOVN**: Move wide with NOT.

**MOVZ**: Move wide with zero.

**MRS**: Move System Register.

**MSR (immediate)**: Move immediate value to Special Register.

**MSR (register)**: Move general-purpose register to System Register.

**MSUB**: Multiply-Subtract.

**MUL**: Multiply: an alias of MADD.

**MVN**: Bitwise NOT: an alias of ORN (shifted register).

**NEG (shifted register)**: Negate (shifted register): an alias of SUB (shifted register).

**NEGS**: Negate, setting flags: an alias of SUBS (shifted register).

**NGC**: Negate with Carry: an alias of SBC.

**NGCS**: Negate with Carry, setting flags: an alias of SBCS.

**NOP**: No Operation.

**ORN (shifted register)**: Bitwise OR NOT (shifted register).

**ORR (immediate)**: Bitwise OR (immediate).

**ORR (shifted register)**: Bitwise OR (shifted register).

**PACDA, PACDZA**: Pointer Authentication Code for Data address, using key A.

**PACDB, PACDBZ**: Pointer Authentication Code for Data address, using key B.

**PACGA**: Pointer Authentication Code, using Generic key.

**PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA**: Pointer Authentication Code for Instruction address, using key A.

**PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB**: Pointer Authentication Code for Instruction address, using key B.

**PRFM (immediate)**: Prefetch Memory (immediate).

**PRFM (literal)**: Prefetch Memory (literal).
**PRFM (register)**: Prefetch Memory (register).

**PRFUM**: Prefetch Memory (unscaled offset).

**PSB CSYNC**: Profiling Synchronization Barrier.

**PSSBB**: Physical Speculative Store Bypass Barrier.

**RBIT**: Reverse Bits.

**RET**: Return from subroutine.

**RETA, RETAB**: Return from subroutine, with pointer authentication.

**REV**: Reverse Bytes.

**REV16**: Reverse bytes in 16-bit halfwords.

**REV32**: Reverse bytes in 32-bit words.

**REV64**: Reverse Bytes: an alias of REV.

**RMIF**: Rotate, Mask Insert Flags.

**ROR (immediate)**: Rotate right (immediate): an alias of EXTR.

**ROR (register)**: Rotate Right (register): an alias of RORV.

**RORV**: Rotate Right Variable.

**SB**: Speculation Barrier.

**SBC**: Subtract with Carry.

**SBCS**: Subtract with Carry, setting flags.

**SBFIZ**: Signed Bitfield Insert in Zero: an alias of SBFM.

**SBFM**: Signed Bitfield Move.

**SBFX**: Signed Bitfield Extract: an alias of SBFM.

**SDIV**: Signed Divide.

**SETF8, SETF16**: Evaluation of 8 or 16 bit flag values.

**SEV**: Send Event.

**SEVL**: Send Event Local.

**SMADDL**: Signed Multiply-Add Long.

**SMC**: Secure Monitor Call.

**SMNEGL**: Signed Multiply-Negate Long: an alias of SMSUBL.

**SMSUBL**: Signed Multiply-Subtract Long.

**SMULH**: Signed Multiply High.

**SMULL**: Signed Multiply Long: an alias of SMADDL.

**SSBB**: Speculative Store Bypass Barrier.

**ST2G**: Store Allocation Tags.

**ST64B**: Single-copy Atomic 64-byte Store without Return.

**ST64BV**: Single-copy Atomic 64-byte Store with Return.

**ST64BV0**: Single-copy Atomic 64-byte EL0 Store with Return.
STADD, STADDL: Atomic add on word or doubleword in memory, without return: an alias of LDADD, LDADDA, LDADDAL, LDADDDL.

STADDB, STADDLB: Atomic add on byte in memory, without return: an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB.

STADDH, STADDLH: Atomic add on halfword in memory, without return: an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH.

STCLR, STCLRL: Atomic bit clear on word or doubleword in memory, without return: an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL.

STCLRB, STCLRLB: Atomic bit clear on byte in memory, without return: an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.

STCLRH, STCLRLH: Atomic bit clear on halfword in memory, without return: an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.

STEOR, STEORL: Atomic exclusive OR on word or doubleword in memory, without return: an alias of LDEOR, LDEORA, LDEORAL, LDEORL.

STEORB, STEORLB: Atomic exclusive OR on byte in memory, without return: an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB.

STEORH, STEORLH: Atomic exclusive OR on halfword in memory, without return: an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH.

STG: Store Allocation Tag.

STGM: Store Tag Multiple.

STGP: Store Allocation Tag and Pair of registers.

STLLR: Store LORelease Register.

STLLRB: Store LORelease Register Byte.

STLLRH: Store LORelease Register Halfword.

STLR: Store-Release Register.

STLRB: Store-Release Register Byte.

STLRH: Store-Release Register Halfword.

STLUR: Store-Release Register (unscaled).

STLURB: Store-Release Register Byte (unscaled).

STLURH: Store-Release Register Halfword (unscaled).

STLXP: Store-Release Exclusive Pair of registers.

STLXR: Store-Release Exclusive Register.

STLXRB: Store-Release Exclusive Register Byte.

STLXRH: Store-Release Exclusive Register Halfword.

STNP: Store Pair of Registers, with non-temporal hint.

STP: Store Pair of Registers.

STR (immediate): Store Register (immediate).

STR (register): Store Register (register).

STRB (immediate): Store Register Byte (immediate).

STRB (register): Store Register Byte (register).

STRH (immediate): Store Register Halfword (immediate).
**STRH (register)**: Store Register Halfword (register).

**STSET, STSETL**: Atomic bit set on word or doubleword in memory, without return: an alias of LDSET, LDSETA, LDSETAL, LDSETL.

**STSETB, STSETLB**: Atomic bit set on byte in memory, without return: an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB.

**STSETH, STSETLH**: Atomic bit set on halfword in memory, without return: an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH.

**STSMAX, STSMAXL**: Atomic signed maximum on word or doubleword in memory, without return: an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.

**STSMAXB, STSMAXLB**: Atomic signed maximum on byte in memory, without return: an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.

**STSMAXH, STSMAXLH**: Atomic signed maximum on halfword in memory, without return: an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.

**STSMIN, STSMINL**: Atomic signed minimum on word or doubleword in memory, without return: an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.

**STSMINB, STSMINLB**: Atomic signed minimum on byte in memory, without return: an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.

**STSMINH, STSMINLH**: Atomic signed minimum on halfword in memory, without return: an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.

**STTR**: Store Register (unprivileged).

**STTRB**: Store Register Byte (unprivileged).

**STTRH**: Store Register Halfword (unprivileged).

**STUMAX, STUMAXL**: Atomic unsigned maximum on word or doubleword in memory, without return: an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.

**STUMAXB, STUMAXLB**: Atomic unsigned maximum on byte in memory, without return: an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.

**STUMAXH, STUMAXLH**: Atomic unsigned maximum on halfword in memory, without return: an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.

**STUMIN, STUMINL**: Atomic unsigned minimum on word or doubleword in memory, without return: an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.

**STUMINB, STUMINLB**: Atomic unsigned minimum on byte in memory, without return: an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.

**STUMINH, STUMINLH**: Atomic unsigned minimum on halfword in memory, without return: an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.

**STUR**: Store Register (unscaled).

**STURB**: Store Register Byte (unscaled).

**STURH**: Store Register Halfword (unscaled).

**STXP**: Store Exclusive Pair of registers.

**STXR**: Store Exclusive Register.

**STXRB**: Store Exclusive Register Byte.

**STXRH**: Store Exclusive Register Halfword.

**STZ2G**: Store Allocation Tags, Zeroing.

**STZG**: Store Allocation Tag, Zeroing.

**STZGM**: Store Tag and Zero Multiple.
**SUB (extended register)**: Subtract (extended register).

**SUB (immediate)**: Subtract (immediate).

**SUB (shifted register)**: Subtract (shifted register).

**SUBG**: Subtract with Tag.

**SUBP**: Subtract Pointer.

**SUBPS**: Subtract Pointer, setting Flags.

**SUBS (extended register)**: Subtract (extended register), setting flags.

**SUBS (immediate)**: Subtract (immediate), setting flags.

**SUBS (shifted register)**: Subtract (shifted register), setting flags.

**SVC**: Supervisor Call.

**SWP, SWPA, SWPAL, SWPL**: Swap word or doubleword in memory.

**SWPB, SWPAB, SWPALB, SWPLB**: Swap byte in memory.

**SWPH, SWPAH, SWPALH, SWPLH**: Swap halfword in memory.

**SXTR**: Signed Extend Byte: an alias of SBFM.

**SXTH**: Sign Extend Halfword: an alias of SBFM.

**SXTW**: Sign Extend Word: an alias of SBFM.

**SYS**: System instruction.

**SYSL**: System instruction with result.

**TBNZ**: Test bit and Branch if Nonzero.

**TBZ**: Test bit and Branch if Zero.

**TCANCEL**: Cancel current transaction.

**TCOMMIT**: Commit current transaction.

**TLBI**: TLB Invalidate operation: an alias of SYS.

**TSB CSYNC**: Trace Synchronization Barrier.

**TST (immediate)**: Test bits (immediate): an alias of ANDS (immediate).

**TST (shifted register)**: Test (shifted register): an alias of ANDS (shifted register).

**TSTART**: Start transaction.

**TTEST**: Test transaction state.

**UBFIZ**: Unsigned Bitfield Insert in Zero: an alias of UBFM.

**UBFM**: Unsigned Bitfield Move.

**UBFX**: Unsigned Bitfield Extract: an alias of UBFM.

**UDF**: Permanently Undefined.

**UDIV**: Unsigned Divide.

**UMADDL**: Unsigned Multiply-Add Long.

**UMNEGL**: Unsigned Multiply-Negate Long: an alias of UMSUBL.

**UMSUBL**: Unsigned Multiply-Subtract Long.
**UMULH**: Unsigned Multiply High.

**UMULL**: Unsigned Multiply Long: an alias of UMADDL.

**UXTB**: Unsigned Extend Byte: an alias of UBFM.

**UXTH**: Unsigned Extend Halfword: an alias of UBFM.

**WFE**: Wait For Event.

**WFET**: Wait For Event with Timeout.

**WFI**: Wait For Interrupt.

**WFIT**: Wait For Interrupt with Timeout.

**XAFLAG**: Convert floating-point condition flags from external format to Arm format.

**XPACD, XPACI, XPACLRI**: Strip Pointer Authentication Code.

**YIELD**: YIELD.
ADC

Add with Carry adds two register values and the Carry flag value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 0 1 0 0 0 0</th>
<th>Rm</th>
<th>0 0 0 0 0 0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADC <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

ADC <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);
X[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADCS

Add with Carry, setting flags, adds two register values and the Carry flag value, and writes the result to the destination register. It updates the condition flags based on the result.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADCS <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

ADCS <Xd>, <Xn>, < Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;
(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  sf 0 0 0 1 0 1 1 0 0 1    Rm  |  op |  option  |  imm3  |  Rn  |  Rd
```

32-bit (sf == 0)

ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

64-bit (sf == 1)

ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler Symbols

- `<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>` Is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th><code>option</code></th>
<th><code>&lt;R&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

- `<m>` Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- `<extend>` For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th><code>option</code></th>
<th><code>&lt;extend&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases `<extend>` is required and must be UXTW when "option" is '010'.
For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
(result, -) = AddWithCarry(operand1, operand2, '0');
if d == 31 then
  SP[] = result;
else
  X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>sh</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADD <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}

64-bit (sf == 1)

ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to/from SP)</td>
<td>sh == '0' &amp;&amp; imm12 == '000000000000' &amp;&amp; (Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then \texttt{SP}[] else \texttt{X}[n];

(result, -) = \texttt{AddWithCarry}(operand1, \texttt{imm}, '0');

if d == 31 then
  \texttt{SP}[] = result;
else
  \texttt{X}[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf  | 0  | 0  | 1  | 0  | 1  | shift | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| op  | 5  |    |    |    |    |        |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

32-bit (sf == 0)

ADD <Wd>, <Wn>, <Wm>{{ <shift> #<amount>}}

64-bit (sf == 1)

ADD <Xd>, <Xn>, <Xm>{{ <shift> #<amount>}}

```plaintext
text
```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

```
<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
(result, -) = AddWithCarry(operand1, operand2, '0');
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ADDG**

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

**Integer (Armv8.5)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | uimm6 | [0](0) | uimm4 | Xn | Xd |

ADDG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>

if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);

**Assembler Symbols**

- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer; encoded in the “Xn” field.
- `<uimm6>` Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
- `<uimm4>` Is an unsigned immediate, in the range 0 to 15, encoded in the “uimm4" field.

**Operation**

bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
  rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
  rtag = '0000';
(result, -) = AddWithCarry(operand1, offset, '0');</p>
result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);
if d == 31 then
  SP[] = result;
else
  X[d] = result;
ADDS (extended register)

Add (extended register), setting flags, adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMN (extended register).

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>imm3</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ADDS <Wd>, <Wn|WSP>, <Wm>{, <extend> {=#<amount>}}

64-bit (sf == 1)

ADDS <Xd>, <Xn|SP>, <R><m>{, <extend> {=#<amount>}}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

ADDS (extended register)
For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
(result, nzcv) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ADDS (immediate)**

Add (immediate), setting flags, adds a register value and an optionally-shifted immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **CMN (immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

```
ADDS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
ADDS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

integer d = UInt(Rd);
n = UInt(Rn);
datasize = if sf == '1' then 64 else 32;

```plaintext
case sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (immediate)</td>
<td>Rd == '1111'</td>
</tr>
</tbody>
</table>

**Operation**

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(4) nzcv;
(result, nzcv) = AddWithCarry(operand1, imm, '0');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDS (shifted register)

Add (shifted register), setting flags, adds a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMN (shifted register).

32-bit (sf == 0)

ADDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

ADDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMN (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

(result, nzcv) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
   • The response of this instruction to asynchronous exceptions does not vary based on:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
ADR

Form PC-relative address adds an immediate value to the PC value to form a PC-relative address, and writes the result to the destination register.

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op|  0 |immlo| 1 | 0 | 0 | 0 | 0 | immhi|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

**Assembler Symbols**

- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<label>` is the program label whose address is to be calculated. Its offset from the address of this instruction, in the range +/-1MB, is encoded in “immhi:immlo”.

**Operation**

```
integer d = UInt(Rd);
bits(64) imm;
imm = SignExtend(immhi:immlo, 64);

bits(64) base = PC[];
X[d] = base + imm;
```
ADRP

Form PC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits, to the PC value to form a PC-relative address, with the bottom 12 bits masked out, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>immlo</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADRP <Xd>, <label>

integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo:Zeros(12), 64);

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<label> Is the program label whose 4KB page address is to be calculated. Its offset from the page address of this instruction, in the range +/-4GB, is encoded as "immhi:immlo" times 4096.

Operation

bits(64) base = PC[];
base<11:0> = Zeros(12);

X[d] = base + imm;

AND (immediate)

Bitwise AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0)

AND <Wd|WSP>, <Wn>, #<imm>

64-bit (sf == 1)

AND <Xd|SP>, <Xn>, #<imm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];

result = operand1 AND imm;
if d == 31 then
    SP[] = result;
else
    X[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Bitwise AND (shifted register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th>sf</th>
<th>0 0</th>
<th>0 1</th>
<th>0 1</th>
<th>shift</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

AND <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

AND <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);
type m = UInt(Rm);
type datatize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
type integer shift_amount = UInt(imm6);
```

Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>`: Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>`: For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

```plaintext
bits(datatize) operand1 = X[n];
bits(datatize) operand2 = ShiftReg(m, shift_type, shift_amount);
result = operand1 AND operand2;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
The values of the data supplied in any of its registers.
The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ANDS (immediate)**

Bitwise AND (immediate), setting flags, performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias **TST (immediate)**.

32-bit ($sf == 0 && N == 0$)

```
ANDS <Wd>, <Wn>, #<imm>
```

64-bit ($sf == 1$)

```
ANDS <Xd>, <Xn>, #<imm>
```

### Assembler Symbols

- **<Wd>**
  
  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

- **<Wn>**
  
  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

- **<Xd>**
  
  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

- **<Xn>**
  
  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

- **<imm>**
  
  For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TST (immediate)</strong></td>
<td>Rd == '1111'</td>
</tr>
</tbody>
</table>

### Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
result = operand1 AND imm;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
X[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
ANDS (shifted register)

Bitwise AND (shifted register), setting flags, performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias TST (shifted register).

32-bit (sf == 0)

ANDS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

ANDS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;
ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>TST (shifted register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

Page 35
Operation

\[
\text{bits(datasize)} \text{ operand1} = X[n];
\]

\[
\text{bits(datasize)} \text{ operand2} = \text{ShiftReg}(m, \text{shift\_type}, \text{shift\_amount});
\]

result = operand1 AND operand2;

PSTATE.<N,Z,C,V> = result<datasize-1>:\text{IsZeroBit(result)}:'00';

\[
X[d] = \text{result};
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of ASRV. This means:

- The encodings in this description are named to match the encodings of ASRV.
- The description of ASRV gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ASR <Wd>, <Wn>, <Wm>

is equivalent to

ASRV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

ASR <Xd>, <Xn>, <Xm>

is equivalent to

ASRV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of ASRV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of the sign bit in the upper bits and zeros in the lower bits, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

### 32-bit (sf == 0 && N == 0 && imms == 011111)

ASR <Wd>, <Wn>, #<shift>

is equivalent to

SBFM <Wd>, <Wn>, #<shift>, #31

and is always the preferred disassembly.

### 64-bit (sf == 1 && N == 1 && imms == 111111)

ASR <Xd>, <Xn>, #<shift>

is equivalent to

SBFM <Xd>, <Xn>, #<shift>, #63

and is always the preferred disassembly.

#### Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <shift> For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

#### Operation

The description of SBFM gives the operational pseudocode for this instruction.

#### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ASRV

Arithmetic Shift Right Variable shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ASR (register).

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

ASRV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

ASRV <Xd>, <Xn>, <Xm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```plaintext
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Address Translate. For more information, see `op0==0b01, cache maintenance, TLB maintenance, and address translation instructions`.

This is an alias of `SYS`. This means:

- The encodings in this description are named to match the encodings of `SYS`.
- The description of `SYS` gives the operational pseudocode for this instruction.

![](image)

AT `<at_op>, <Xt>`

is equivalent to

`SYS #<op1>, C7, <Cm>, #<op2>, <Xt>`

and is the preferred disassembly when `SysOp(op1, '0111', CRm, op2) == Sys_AT`.

### Assembler Symbols

`<at_op>`

Is an AT instruction name, as listed for the AT system instruction group, encoded in "op1:CRm<0>:op2":

<table>
<thead>
<tr>
<th><code>op1</code></th>
<th><code>CRm&lt;0&gt;</code></th>
<th><code>op2</code></th>
<th><code>&lt;at_op&gt;</code></th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>000</td>
<td>S1E1R</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>001</td>
<td>S1E1W</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>010</td>
<td>S1E0R</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
<td>011</td>
<td>S1E0W</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>000</td>
<td>S1E1RP</td>
<td>FEAT_PAN2</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
<td>001</td>
<td>S1E1WP</td>
<td>FEAT_PAN2</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>000</td>
<td>S1E2R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>001</td>
<td>S1E2W</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>100</td>
<td>S1E1R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>101</td>
<td>S1E1W</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>110</td>
<td>S1E0R</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>111</td>
<td>S1E0W</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>000</td>
<td>S1E3R</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
<td>001</td>
<td>S1E3W</td>
<td>-</td>
</tr>
</tbody>
</table>

`<op1>`

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

`<Cm>`

Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

`<op2>`

Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

`<Xt>`

Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

### Operation

The description of `SYS` gives the operational pseudocode for this instruction.
AUTDA, AUTDZA

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A. The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDA.
- The value zero, for AUTDZA.

If the authentication fails, the upper bits of the address are restored to enable subsequent use of the address. If the authentication passes, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

**Integer (Armv8.3)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | Rn | Rd |

**AUTDA (Z == 0)**

AUTDA <Xd>, <Xn|SP>

**AUTDZA (Z == 1 && Rn == 11111)**

AUTDZA <Xd>

```plaintext
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // AUTDA
    if n == 31 then source_is_sp = TRUE;
    else // AUTDZA
        if n != 31 then UNDEFINED;
```

**Assembler Symbols**

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
if HavePACExt() then
    if source_is_sp then
        X[d] = AuthDA(X[d], SP[], FALSE);
    else
        X[d] = AuthDA(X[d], X[n], FALSE);
```
AUTDB, AUTDZB

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B. The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTDB.
- The value zero, for AUTDZB.

If the authentication fails, the upper bits of the address are corrupted and any subsequent use of the address results in a Translation fault.

Integer
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | Rn |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

AUTDB (Z == 0)

AUTDB <Xd>, <Xn|SP>

AUTDZB (Z == 1 && Rn == 11111)

AUTDZB <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // AUTDB
    if n == 31 then source_is_sp = TRUE;
else // AUTDZB
    if n != 31 then UNDEFINED;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

if HavePACExt() then
    if source_is_sp then
        X[d] = AuthDB(X[d], SP[], FALSE);
    else
        X[d] = AuthDB(X[d], X[n], FALSE);
**AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA**

Authenticate Instruction address, using key A. This instruction authenticates an instruction address, using a modifier and key A.

The address is:
- In the general-purpose register that is specified by <Xd> for AUTIA and AUTIZA.
- In X17, for AUTIA1716.
- In X30, for AUTIASP and AUTIAZ.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIA.
- The value zero, for AUTIZA and AUTIAZ.
- In X16, for AUTIA1716.
- In SP, for AUTIASP.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: **Integer** and **System**

**Integer**
(Armv8.3)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 1  | 0  | 0  | Rn | Rd |
```

**AUTIA (Z == 0)**

AUTIA <Xd>, <Xn|SP>

**AUTIZA (Z == 1 && Rn == 11111)**

AUTIZA <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // AUTIA
    if n == 31 then source_is_sp = TRUE;
else // AUTIZA
    if n != 31 then UNDEFINED;
```

**System**
(Armv8.3)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | x  | 1  | 1  | 0  | x  | 1  | 1  | 1  | 1  | 1  |
```

AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA
AUTIA1716 (CRm == 0001 && op2 == 100)

AUTIA1716

AUTIASP (CRm == 0011 && op2 == 101)

AUTIASP

AUTIAZ (CRm == 0011 && op2 == 100)

AUTIAZ

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 100'    // AUTIAZ
        d = 30;
        n = 31;
    when '0011 101'    // AUTIASP
        d = 30;
        source_is_sp = TRUE;
    when '0001 100'    // AUTIA1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 010' SEE "PACIB";
    when '0001 110' SEE "AUTIB";
    when '0011 00x' SEE "PACIA";
    when '0011 01x' SEE "PACIB";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt() then
    if source_is_sp then
        \texttt{X}[d] = \texttt{AuthIA}(\texttt{X}[d], SP[], FALSE);
    else
        \texttt{X}[d] = \texttt{AuthIA}(\texttt{X}[d], X[n], FALSE);
**AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB**

Authenticate Instruction address, using key B. This instruction authenticates an instruction address, using a modifier and key B.

The address is:
- In the general-purpose register that is specified by \( <Xd> \) for **AUTIB** and **AUTIZB**.
- In X17, for **AUTIB1716**.
- In X30, for **AUTIBSP** and **AUTIBZ**.

The modifier is:
- In the general-purpose register or stack pointer that is specified by \( <Xn|SP> \) for **AUTIB**.
- The value zero, for **AUTIZB** and **AUTIBZ**.
- In X16, for **AUTIB1716**.
- In SP, for **AUTIBSP**.

If the authentication passes, the upper bits of the address are restored to enable subsequent use of the address. If the authentication fails, the upper bits are corrupted and any subsequent use of the address results in a Translation fault.

It has encodings from 2 classes: **Integer** and **System**

### Integer (Armv8.3)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 1  | 0  | 1  | Rn | Rd |
```

**AUTIB (Z == 0)**

**AUTIB <Xd>, <Xn|SP>**

**AUTIZB (Z == 1 && Rn == 1111)**

**AUTIZB <Xd>**

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
  UNDEFINED;

if Z == '0' then // AUTIB
  if n == 31 then source_is_sp = TRUE;
else // AUTIZB
  if n != 31 then UNDEFINED;
```

### System (Armv8.3)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | x  | 1  | 1  | 1  | x  | 1  | 1  | 1  | 1  | 1  |
```

**CRm**

**op2**
AUTIB1716 (CRm == 0001 && op2 == 110)

AUTIB1716

AUTIBSP (CRm == 0011 && op2 == 111)

AUTIBSP

AUTIBZ (CRm == 0011 && op2 == 110)

AUTIBZ

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 110'    // AUTIBZ
    d = 30;
    n = 31;
  when '0011 111'    // AUTIBSP
    d = 30;
    source_is_sp = TRUE;
  when '0001 110'    // AUTIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 010' SEE "PACIB";
  when '0001 100' SEE "AUTIA";
  when '0011 00x' SEE "PACIA";
  when '0011 01x' SEE "PACIB";
  when '0011 10x' SEE "AUTIA";
  when '0000 111' SEE "XPACLRI";
  otherwise SEE "HINT";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt() then
  if source_is_sp then
    X[d] = AuthIB(X[d], SP[], FALSE);
  else
    X[d] = AuthIB(X[d], X[n], FALSE);

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AXFLAG

Convert floating-point condition flags from Arm to external format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an Arm floating-point scalar compare instruction to an alternative representation required by some software.

System (Armv8.5)

```
0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1
```

```plaintext
AXFLAG

if !HaveFlagFormatExt() then UNDEFINED;

Operation

bit Z = PSTATE.Z OR PSTATE.V;
bite C = PSTATE.C AND NOT(PSTATE.V);

PSTATE.N = '0';
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = '0';
```
B.cond

Branch conditionally to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

B.<cond> <label>

bits(64) offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

if ConditionHolds(cond) then
    BranchTo(PC[] + offset, BranchType_DIR);

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

```
0 | 0 0 0 1 0 1 |
---|-------------|
imm26
```

**Assembler Symbols**

```<label>`
Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as “imm26" times 4.

**Operation**

```
BranchTo(PC[] + offset, BranchType_DIR);
```
BFC

Bitfield Clear sets a bitfield of <width> bits at bit position <lsb> of the destination register to zero, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

Leaving other bits unchanged
(Armv8.2)

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>Nimmr</td>
<td>immr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0)

BFC <Wd>, #<lsb>, #<width>

is equivalent to

BFM <Wd>, WZR, #(-<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 && N == 1)

BFC <Xd>, #<lsb>, #<width>

is equivalent to

BFM <Xd>, XZR, #(-<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <lsb> For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- <width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BFI

Bitfield Insert copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 & N == 0)

BFI <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

BFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 & N == 1)

BFI <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

BFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<lsb> For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.

For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.

<width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.

For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If \(<imm>\) is greater than or equal to \(<imm>\), this copies a bitfield of \((<imm>-<imm>)+1\) bits starting from bit position \(<imm>\) in the source register to the least significant bits of the destination register.

If \(<imm>\) is less than \(<imm>\), this copies a bitfield of \((<imm>+1\) bits from the least significant bits of the source register to bit position \((regsize-<imm>)\) of the destination register, where \(regsize\) is the destination register size of 32 or 64 bits.

In both cases the other bits of the destination register remain unchanged.

This instruction is used by the aliases BFC, BFI, and BFXIL.

###Assembler Symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<imm>\) For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "imm" field.
  - For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "imm" field.
- \(<imms>\) For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field.
  - For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

###Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFC</td>
<td>(Rn == '11111' &amp;&amp; \text{\texttt{UInt}}(imms) &lt; \text{\texttt{UInt}}(immr))</td>
</tr>
<tr>
<td>BFI</td>
<td>(Rn != '11111' &amp;&amp; \text{\texttt{UInt}}(imms) &lt; \text{\texttt{UInt}}(immr))</td>
</tr>
<tr>
<td>BFXIL</td>
<td>(\text{\texttt{UInt}}(imms) &gt;= \text{\texttt{UInt}}(immr))</td>
</tr>
</tbody>
</table>
**Operation**

bits(datasize) dst = X[d];
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = (dst AND NOT(wmask)) OR (ROR(src, R) AND wmask);

// combine extension bits and result bits
X[d] = (dst AND NOT(tmask)) OR (bot AND tmask);

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BFXIL

Bitfield Extract and Insert Low copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

- The encodings in this description are named to match the encodings of BFM.
- The description of BFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0)

BFXIL <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

BFM <Wd>, <Wn>, #<lsb>, #{<lsb>+<width>-1}

and is the preferred disassembly when UInt(imms) >= UInt(immr).

64-bit (sf == 1 && N == 1)

BFXIL <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

BFM <Xd>, <Xn>, #<lsb>, #{<lsb>+<width>-1}

and is the preferred disassembly when UInt(imms) >= UInt(immr).

Assembler Symbols

- **<Wd>**: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>**: Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<Xd>**: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>**: Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- **<lsb>**: For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- **<width>**: For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of BFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**BIC (shifted register)**

Bitwise Bit Clear (shifted register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

BIC <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

BIC <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  - For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
operand2 = NOT(operand2);
result = operand1 AND operand2;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BICS (shifted register)

Bitwise Bit Clear (shifted register), setting flags, performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

BICS `<Wd>, <Wn>, <Wm>{, <shift> #<amount>}`

**64-bit (sf == 1)**

BICS `<Xd>, <Xn>, <Xm>{, <shift> #<amount>}`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINE;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

**Operation**

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
operand2 = NOT(operand2);
result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';
X[d] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 | 0 0 1 0 1 | imm26
```

```
BL <label>

bits(64) offset = SignExtend(imm26:'00', 64);
```

**Assembler Symbols**

<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as “imm26” times 4.

**Operation**

```
if HaveCSRExt() && CallStackRecordingActive() then
    AddCallStackRecord();
X[30] = PC[] + 4;
BranchTo(PC[] + offset, BranchType_DIRCALL);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
```

BLR `<Xn>`

integer n = UInt(Rn);

**Assembler Symbols**

`<Xn>` Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field.

**Operation**

```
bits(64) target = X[n];

if HaveCSRExt() && CallStackRecordingActive() then
    AddCallStackRecord();
    X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ.
If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Integer (Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 1  | Z | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | M | Rn | Rm |

Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

BLRAAZ <Xn>

Key A, register modifier (Z == 1 && M == 0)

BLRAA <Xn>, <Xm|SP>

Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

BLRABZ <Xn>

Key B, register modifier (Z == 1 && M == 1)

BLRAB <Xn>, <Xm|SP>

```java
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !HavePACExt() then
    UNDEFINED;
if Z == '0' && m != 31 then
    UNDEFINED;
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.
Operation

bits(64) target = X[n];

bits(64) modifier = if source_is_sp then SP[] else X[m];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

if HaveCSRExt() && CallStackRecordingActive() then
    AddCallStackRecord();
X[30] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

```
BR <Xn>

integer n = UInt(Rn);
```

**Assembler Symbols**

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

**Operation**

```
bits(64) target = X[n];

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
  if n == 16 || n == 17 then
    BTypeNext = '01';
  else
    BTypeNext = '11';
else
  BTypeNext = '01';
BranchTo(target, BranchType_INDIR);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and branches to the authenticated address. The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Integer
(ARMv8.3)

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| op | 1  | 1  | 0  | 1  | 1  | Z | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | M | Rn | Rm | A |

Key A, zero modifier (Z == 0 & M == 0 & Rm == 11111)

BRAAZ <Xn>

Key A, register modifier (Z == 1 & M == 0)

BRAA <Xn>, <Xm|SP>

Key B, zero modifier (Z == 0 & M == 1 & Rm == 11111)

BRABZ <Xn>

Key B, register modifier (Z == 1 & M == 1)

BRAB <Xn>, <Xm|SP>

```java
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = (Z == '1') && (m == 31);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' && m != 31 then
    UNDEFINED;
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

<Xm|SP> Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.
Operation

bits(64) target = X[n];

bits(64) modifier = if source_is_sp then SP[] else X[m];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';
BranchTo(target, BranchType_INDIR);
Breakpoint instruction. A BRK instruction generates a Breakpoint Instruction exception. The PE records the exception in ESR_ELx, using the EC value 0x3c, and captures the value of the immediate argument in ESR_ELx.ISS.

BRK #<imm>

if HaveBTIExt() then
    SetBTypeCompatible(TRUE);

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

AArch64.SoftwareBreakpoint(imm16);
BTI

Branch Target Identification. A BTI instruction is used to guard against the execution of instructions which are not the intended target of a branch.

Outside of a guarded memory region, a BTI instruction executes as a NOP. Within a guarded memory region while PSTATE.BTYPE != 0b00, a BTI instruction compatible with the current value of PSTATE.BTYPE will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region.

The operand <targets> passed to a BTI instruction determines the values of PSTATE.BTYPE which the BTI instruction is compatible with.

Within a guarded memory region, while PSTATE.BTYPE != 0b00, all instructions will generate a Branch Target Exception, other than BRK, BTI, HLT, PACIASP, and PACIBSP, which may not. See the individual instructions for details.

System
(Armv8.5)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 | 0 0 0 1 1 0 0 1 0 | 0 1 0 0 | x x 0 | 1 1 1 1 1 1
CRm | op2
```

```cpp
BTI {<targets>}

SystemHintOp op;

if CRm:op2 == '0100 xx0' then
    op = SystemHintOp_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
else
    EndOfInstruction();
```

Assembler Symbols

<targets> Is the type of indirection, encoded in “op2<2:1>”:

<table>
<thead>
<tr>
<th>op2&lt;2:1&gt;</th>
<th>&lt;targets&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(omitted)</td>
</tr>
<tr>
<td>01</td>
<td>c</td>
</tr>
<tr>
<td>10</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>jc</td>
</tr>
</tbody>
</table>
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_DGH
    Hint_DGH();
  when SystemHintOp_WFE
    Hint_WFE(-1);
  when SystemHintOp_WFI
    Hint_WFI(-1);
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.vESB0peration();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
      AArch64.vESB0peration();
      TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    SetBTypeNext('00');
  otherwise    // do nothing
CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASA and CASAL load from memory with acquire semantics.
- CASL and CASAL store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*

For information about memory accesses see *Load/Store addressing modes.*

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

### No offset

*(Armv8.1)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 0  | 1  | 0  | 0  | 1  | L  | 1  | Rs | 00 | 1  | 1  | 1  | 1  | 1  | Rn | |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
</table>

*No offset (Armv8.1)*
32-bit CAS: \(size == 10 \land L == 0 \land o0 == 0\)

CAS \(<Ws>, <Wt>, [<Xn|SP>{,#0}]\)

32-bit CASA: \(size == 10 \land L == 1 \land o0 == 0\)

CASA \(<Ws>, <Wt>, [<Xn|SP>{,#0}]\)

32-bit CASAL: \(size == 10 \land L == 1 \land o0 == 1\)

CASAL \(<Ws>, <Wt>, [<Xn|SP>{,#0}]\)

32-bit CASL: \(size == 10 \land L == 0 \land o0 == 1\)

CASL \(<Ws>, <Wt>, [<Xn|SP>{,#0}]\)

64-bit CAS: \(size == 11 \land L == 0 \land o0 == 0\)

CAS \(<Xs>, <Xt>, [<Xn|SP>{,#0}]\)

64-bit CASA: \(size == 11 \land L == 1 \land o0 == 0\)

CASA \(<Xs>, <Xt>, [<Xn|SP>{,#0}]\)

64-bit CASAL: \(size == 11 \land L == 1 \land o0 == 1\)

CASAL \(<Xs>, <Xt>, [<Xn|SP>{,#0}]\)

64-bit CASL: \(size == 11 \land L == 0 \land o0 == 1\)

CASL \(<Xs>, <Xt>, [<Xn|SP>{,#0}]\)

\[
\text{if } \neg \text{ HaveAtomicExt }() \text{ then UNDEFINED;}
\]

\[
\begin{align*}
\text{integer } n &= \text{ UInt}(Rn); \\
\text{integer } t &= \text{ UInt}(Rt); \\
\text{integer } s &= \text{ UInt}(Rs); \\
\text{integer } \text{ datasize} &= 8 \left<\text{ UInt}(\text{ size})\right> \text{ if datarsize == 64 then 64 else 32;}
\end{align*}
\]

\[
\begin{align*}
\text{AccType } \text{ ldacctype } &= \text{ if } L == \text{ '1'} \text{ then AccType ORDEREDATOMICRW else AccType ATOMICRW;}
\text{AccType } \text{ stacctype } &= \text{ if } o0 == \text{ '1'} \text{ then AccType ORDEREDATOMICRW else AccType ATOMICRW;}
\text{boolean } \text{ tag_checked } &= n != 31;
\end{align*}
\]

**Assembler Symbols**

\(<Ws>\) Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

\(<Wt>\) Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

\(<Xs>\) Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

\(<Xt>\) Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(datasize) comparevalue;
bits(datasize) newvalue;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);
X[s] = ZeroExtend(data, regsize);
CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

**No offset**
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 0  | 0  | 1  | L | 1  | Rs | o0 | 1  | 1  | 1  | 1  | Rn |  |  |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

X[s] = ZeroExtend(data, 32);
CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

**No offset**

(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | L  | 1  | Rs | o0 | 1  | 1  | 1  | 1  | 0  | Rn | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  |
| size |

**CASAH** (L == 1 && o0 == 0)

CASAH <Ws>, <Wt>, [<Xn|SP>{,#0}]

**CASALH** (L == 1 && o0 == 1)

CASALH <Ws>, <Wt>, [<Xn|SP>{,#0}]

**CASH** (L == 0 && o0 == 0)

CASH <Ws>, <Wt>, [<Xn|SP>{,#0}]

**CASLH** (L == 0 && o0 == 1)

CASLH <Ws>, <Wt>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);

```
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

**Assembler Symbols**

- <Ws> Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s];
newvalue = X[t];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

X[s] = ZeroExtend(data, 32);
CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- CASPA and CASPAL load from memory with acquire semantics.
- CASPL and CASPAL store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

No offset
(Armv8.1)

```
0 sz 0 0 1 0 0 0 L 1 Rs 00 1 1 1 1 1 Rn Rt Rt2
```
32-bit CASP (sz == 0 && L == 0 & o0 == 0)
CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPA (sz == 0 && L == 1 & o0 == 0)
CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPAL (sz == 0 && L == 1 & o0 == 1)
CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPL (sz == 0 && L == 0 & o0 == 1)
CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]

64-bit CASP (sz == 1 && L == 0 & o0 == 0)
CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPA (sz == 1 && L == 1 & o0 == 0)
CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPAL (sz == 1 && L == 1 & o0 == 1)
CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPL (sz == 1 && L == 0 & o0 == 1)
CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
integer datasize = 32 << UInt(sz);

AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.

<W(s+1)> Is the 32-bit name of the second general-purpose register to be compared and loaded.

<Wt> Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.

<W(t+1)> Is the 32-bit name of the second general-purpose register to be conditionally stored.

<Xs> Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.

<X(s+1)> Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt> Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.
Is the 64-bit name of the second general-purpose register to be conditionally stored.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s];
bits(datasize) s2 = X[s+1];
bits(datasize) t1 = X[t];
bits(datasize) t2 = X[t+1];
comparevalue = if BigEndian(ldacctype) then s1:s2 else s2:s1;
newvalue = if BigEndian(stacctype) then t1:t2 else t2:t1;
```

```plaintext
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
fi

if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
else
    address = X[n];
```

```plaintext
data = MemAtomicCompareAndSwap(address, comparevalue, newvalue, ldacctype, stacctype);

if BigEndian(ldacctype) then
    X[s] = data<2*datasize-1:datasize>;
    X[s+1] = data<datasize-1:0>;
else
    X[s] = data<datasize-1:0>;
    X[s+1] = data<2*datasize-1:datasize>;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.

<table>
<thead>
<tr>
<th>sf</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CBNZ <Wt>, <label>

64-bit (sf == 1)

CBNZ <Xt>, <label>

    integer t = UInt(Rt);
    integer datasize = if sf == '1' then 64 else 32;
    bits(64) offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

Operation

bits(datasize) operand1 = X[t];

if IsZero(operand1) == FALSE then
    BranchTo(PC[] + offset, BranchType_DIR);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

### 32-bit (sf == 0)

CBZ <Wt>, <label>

### 64-bit (sf == 1)

CBZ <Xt>, <label>

integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);

### Assembler Symbols

- **<Wt>** Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- **<Xt>** Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
- **<label>** Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as “imm19” times 4.

### Operation

bits(datasize) operand1 = X[t];

if IsZero(operand1) == TRUE then
    BranchTo(PC[] + offset, BranchType_DIR);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Conditional Compare Negative (immediate) sets the value of the condition flags to the result of the comparison of a register value and a negated immediate value if the condition is TRUE, and an immediate value otherwise.

### 32-bit (sf == 0)

```
CCMN <Wn>, #<imm>, #<nzcv>, <cond>
```

### 64-bit (sf == 1)

```
CCMN <Xn>, #<imm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

### Assembler Symbols

- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<imm>`: Is a five bit unsigned (positive) immediate encoded in the "imm5" field.
- `<nzcv>`: Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
- `<cond>`: Is one of the standard conditions, encoded in the "cond" field in the standard way.

### Operation

```
bits(datasize) operand1 = X[n];
if ConditionHolds(cond) then
    (\-, flags) = AddWithCarry(operand1, imm, '0');
PSTATE.<N,Z,C,V> = flags;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CCMN (register)

Conditional Compare Negative (register) sets the value of the condition flags to the result of the comparison of a register value and the inverse of another register value if the condition is TRUE, and an immediate value otherwise.

32-bit (sf == 0)

CCMN <Wn>, <Wm>, #<nzcv>, <cond>

64-bit (sf == 1)

CCMN <Xn>, < Xm>, #<nzcv>, <cond>

integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;

Assembler Symbols

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the “nzcv” field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
if ConditionHolds(cond) then
  (,-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CCMP (immediate)

Conditional Compare (immediate) sets the value of the condition flags to the result of the comparison of a register value and an immediate value if the condition is TRUE, and an immediate value otherwise.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>imm5</th>
<th>cond</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>0</th>
<th>nzcv</th>
</tr>
</thead>
</table>

**32-bit (sf == 0)**

CCMP <Wn>, #<imm>, #<nzcv>, <cond>

**64-bit (sf == 1)**

CCMP <Xn>, #<imm>, #<nzcv>, <cond>

integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);

Assembler Symbols

- `<Wn>` is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xn>` is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<imm>` is a five bit unsigned (positive) immediate encoded in the "imm5" field.
- `<nzcv>` is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
- `<cond>` is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2;
if ConditionHolds(cond) then
    operand2 = NOT(imm);
    (\-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CCMP (register)

Conditional Compare (register) sets the value of the condition flags to the result of the comparison of two registers if the condition is TRUE, and an immediate value otherwise.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>0</th>
<th>nzcv</th>
</tr>
</thead>
</table>

32-bit (sf == 0)

CCMP <Wn>, <Wm>, #<nzcv>, <cond>

64-bit (sf == 1)

CCMP <Xn>, <Xm>, #<nzcv>, <cond>

integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;

Assembler Symbols

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcv> Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
if ConditionHolds(cond) then
    operand2 = NOT(operand2);
    (-, flags) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = flags;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CFINV

Invert Carry Flag. This instruction inverts the value of the PSTATE.C flag.

System
(Armv8.4)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | (0)| (0)| (0)| (0)| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  |
```

CFINV

if !HaveFlagManipulateExt() then UNDEFINED;

Operation

PSTATE.C = NOT(PSTATE.C);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses, based on information gathered from earlier execution within a particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see *CFP RCTX, Control Flow Prediction Restriction by Context*.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

**System**  
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |

**CFP RCTX, <Xt>**

is equivalent to

**SYS #3, C7, C3, #4, <Xt>**

and is always the preferred disassembly.

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

**Operation**

The description of SYS gives the operational pseudocode for this instruction.
CINC

Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 0 1 0 0</th>
<th>!= 11111</th>
<th>!= 111x</th>
<th>0 1</th>
<th>!= 11111</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>Rm</td>
<td>cond</td>
<td>o2</td>
<td>Rn</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CINC <Wd>, <Wn>, <cond>

is equivalent to

CSINC <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

CINC <Xd>, <Xn>, <cond>

is equivalent to

CSINC <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.

<cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CINV

Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>!= 11111</th>
<th>!= 1111x</th>
<th>0</th>
<th>0</th>
<th>!= 11111</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>Rm</td>
<td>cond</td>
<td>o2</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CINV <Wd>, <Wn>, <cond>

is equivalent to

CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1)

CINV <Xd>, <Xn>, <cond>

is equivalent to

CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Clear Exclusive clears the local monitor of the executing PE.

```
CLREX {#<imm>}

// CRm field is ignored
```

**Assembler Symbols**

- `<imm>`: Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

**Operation**

```
ClearExclusiveLocal(ProcessorID());
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CLS

Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.

```

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|sf| 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  |

Rn | Rd
---|---

op
```

32-bit (sf == 0)

CLS <Wd>, <Wn>

64-bit (sf == 1)

CLS <Xd>, <Xn>

```plaintext

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```plaintext

integer result;
bits(datasize) operand1 = X[n];
result = CountLeadingSignBits(operand1);
X[d] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CLZ

Count Leading Zeros counts the number of binary zero bits before the first binary one bit in the value of the source register, and writes the result to the destination register.

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Rd |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

32-bit (sf == 0)

CLZ <Wd>, <Wn>

64-bit (sf == 1)

CLZ <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

integer result;
bits(datasize) operand1 = X[n];
result = CountLeadingZeroBits(operand1);
X[d] = result<datasize-1:0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMN (extended register)

Compare Negative (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (extended register). This means:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMN <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

is equivalent to

ADDS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn|SP>, <R><m>{, <extend> {#<amount>}}

is equivalent to

ADDS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>01x</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:
If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMN (immediate)

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (immediate). This means:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CMN <Wn|WSP>, #<imm>{, <shift>}

is equivalent to

ADDS WZR, <Wn|WSP>, #<imm> {, <shift>}

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn|SP>, #<imm>{, <shift>}

is equivalent to

ADDS XZR, <Xn|SP>, #<imm> {, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Wn|WSP> Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMN (shifted register)

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CMN <Wn>, <Wm>{, <shift> #<amount>}

is equivalent to

ADDS WZR, <Wn>, <Wm> {, <shift> #<amount>}

and is always the preferred disassembly.

64-bit (sf == 1)

CMN <Xn>, <Xm>{, <shift> #<amount>}

is equivalent to

ADDS XZR, <Xn>, <Xm> {, <shift> #<amount>}

and is always the preferred disassembly.

Assembler Symbols

- <Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- <shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- <amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**CMP (extended register)**

Compare (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result, and discards the result.

This is an alias of **SUBS (extended register)**. This means:

- The encodings in this description are named to match the encodings of **SUBS (extended register)**.
- The description of **SUBS (extended register)** gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

**CMP** `<Wn|WSP>, <Wm>{, <extend> {#<amount>}}`

is equivalent to

**SUBS** WZR, `<Wn|WSP>, <Wm>{, <extend> {#<amount>}}`

and is always the preferred disassembly.

### 64-bit (sf == 1)

**CMP** `<Xn|SP>, <R><m>{, <extend> {#<amount>}}`

is equivalent to

**SUBS** XZR, `<Xn|SP>, <R><m>{, <extend> {#<amount>}}`

and is always the preferred disassembly.

### Assembler Symbols

- `<Wn|WSP>` Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xn|SP>` Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>` Is a width specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>

- `<m>` Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- `<extend>` For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":

---

**CMP (extended register)**

Page 101
If “Rn” is ‘11111’ (WSP) and “option” is ‘010’ then LSL is preferred, but may be omitted when “imm3” is ‘000’. In all other cases <extend> is required and must be UXTW when "option" is ‘010’.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

<table>
<thead>
<tr>
<th>Option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTB</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTX</td>
</tr>
<tr>
<td>100</td>
<td>SXTH</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If “Rn” is ‘11111’ (SP) and “option” is ‘011’ then LSL is preferred, but may be omitted when "imm3" is ‘000’. In all other cases <extend> is required and must be UXTX when "option" is ‘011’.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMP (immediate)

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (immediate). This means:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

```plaintext
CMP <Wn|WSP>, #<imm>{, <shift>}
```

is equivalent to

```plaintext
SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

### 64-bit (sf == 1)

```plaintext
CMP <Xn|SP>, #<imm>{, <shift>}
```

is equivalent to

```plaintext
SUBS XZR, <Xn|SP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wn|WSP>`: Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xn|SP>`: Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>`: Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

**Operation**

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**CMP (shifted register)**

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of **SUBS (shifted register)**. This means:

- The encodings in this description are named to match the encodings of **SUBS (shifted register)**.
- The description of **SUBS (shifted register)** gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 1 1 0 1 0 1 1 | shift 0 | Rm | imm6 | Rn | 1 1 1 1 1
```

**32-bit (sf == 0)**

\[
\text{CMP } \langle Wn \rangle, \langle Wm \rangle\{, \langle shift \rangle \#\langle amount \rangle\}
\]

is equivalent to

\[
\text{SUBS WZR, } \langle Wn \rangle, \langle Wm \rangle \{, \langle shift \rangle \#\langle amount \rangle\}
\]

and is always the preferred disassembly.

**64-bit (sf == 1)**

\[
\text{CMP } \langle Xn \rangle, \langle Xm \rangle\{, \langle shift \rangle \#\langle amount \rangle\}
\]

is equivalent to

\[
\text{SUBS XZR, } \langle Xn \rangle, \langle Xm \rangle \{, \langle shift \rangle \#\langle amount \rangle\}
\]

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":
  
<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

The description of **SUBS (shifted register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
CMPP

Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This is an alias of SUBPS. This means:

- The encodings in this description are named to match the encodings of SUBPS.
- The description of SUBPS gives the operational pseudocode for this instruction.

**Integer**

(Armv8.5)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CMPP <Xn|SP>, <Xm|SP>

is equivalent to

SUBPS XZR, <Xn|SP>, <Xm|SP>

and is always the preferred disassembly.

**Assembler Symbols**

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the “Xn” field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the “Xm” field.

**Operation**

The description of SUBPS gives the operational pseudocode for this instruction.
CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG. This means:

- The encodings in this description are named to match the encodings of CSNEG.
- The description of CSNEG gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CNEG \(<W_d>, <W_n>, <\text{cond}>\)

is equivalent to

CSNEG \(<W_d>, <W_n>, <W_n>, \text{invert}(<\text{cond}>)\)

and is the preferred disassembly when \(R_n == R_m\).

64-bit (sf == 1)

CNEG \(<X_d>, <X_n>, <\text{cond}>\)

is equivalent to

CSNEG \(<X_d>, <X_n>, <X_n>, \text{invert}(<\text{cond}>)\)

and is the preferred disassembly when \(R_n == R_m\).

Assembler Symbols

- \(<W_d>\) is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<W_n>\) is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- \(<X_d>\) is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<X_n>\) is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- \(<\text{cond}>\) is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSNEG gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions, based on information gathered from earlier execution within a particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see **CPP RCTX, Cache Prefetch Prediction Restriction by Context**.

This is an alias of **SYS**. This means:

- The encodings in this description are named to match the encodings of **SYS**.
- The description of **SYS** gives the operational pseudocode for this instruction.

**System**

(ARMv8.5)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 1 1 1 1 1 1 1 0

L  op1  CRn  CRm  op2  Rt
```

**CPP RCTX, <Xt>**

is equivalent to

**SYS #3, C7, C3, #7, <Xt>**

and is always the preferred disassembly.

**Assembler Symbols**

**<Xt>** Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

**Operation**

The description of **SYS** gives the operational pseudocode for this instruction.
CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.

In Armv8-A, this is an OPTIONAL instruction, and in Armv8.1 it is mandatory for all implementations to implement it. ID AA64ISAR0_EL1. CRC32 indicates whether this instruction is supported.

<table>
<thead>
<tr>
<th>sf</th>
<th>Rm</th>
<th>sz</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

CRC32B (sf == 0 & sz == 00)

CRC32B <Wd>, <Wn>, <Wm>

CRC32H (sf == 0 & sz == 01)

CRC32H <Wd>, <Wn>, <Wm>

CRC32W (sf == 0 & sz == 10)

CRC32W <Wd>, <Wn>, <Wm>

CRC32X (sf == 1 & sz == 11)

CRC32X <Wd>, <Wn>, <Xm>

if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' & sz != '11' then UNDEFINED;
if sf == '0' & sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm> Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

bits(32) acc = X[n];  // accumulator
bits(size) val = X[m];  // input value
bits(32) poly = 0x04C11DB7<31:0>;
bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.

In Armv8-A, this is an **OPTIONAL** instruction, and in Armv8.1 it is mandatory for all implementations to implement it. 

**ID_AA64ISAR0_EL1**. CRC32 indicates whether this instruction is supported.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  |

CRC32CB (sf == 0 & sz == 00)

CRC32CB <Wd>, <Wn>, <Wm>

CRC32CH (sf == 0 & sz == 01)

CRC32CH <Wd>, <Wn>, <Wm>

CRC32CW (sf == 0 & sz == 10)

CRC32CW <Wd>, <Wn>, <Wm>

CRC32CX (sf == 1 & sz == 11)

CRC32CX <Wd>, <Wn>, <Xm>

if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' & sz != '11' then UNDEFINED;
if sf == '0' & sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);
```

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
- **<Wm>** Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

**Operation**

```plaintext
bits(32) acc = X[n];  // accumulator
bits(size) val = X[m];  // input value
bits(32) poly = 0x1EDC6F41<31:0>;

bits(32+size) tempacc = BitReverse(acc) ⊕ Zeros(size);
bits(size+32) tempval = BitReverse(val) ⊕ Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d] = BitReverse(Poly32Mod2(tempacc XOR tempval, poly));
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 1 0 0 0 0 1 1 0 0 1 0 0 0 1 0 1 0 1 0 0 1 1 1 1</td>
</tr>
</tbody>
</table>

CRm   op2
```

// Empty.

### Operation

```
ConsumptionOfSpeculativeDataBarrier();
```
CSEL

If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 0 1 0 0</th>
<th>Rm</th>
<th>cond</th>
<th>0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

**CSEL** `<Wd>`, `<Wn>`, `<Wm>`, `<cond>`

64-bit (sf == 1)

**CSEL** `<Xd>`, `<Xn>`, `<Xm>`, `<cond>`

integer d = `UInt`(Rd);
integer n = `UInt`(Rn);
integer m = `UInt`(Rm);
integer datasize = if sf == '1' then 64 else 32;

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.

**Operation**

```
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds (cond) then
   result = operand1;
else
   result = operand2;

X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSET

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This is an alias of CSINC. This means:

- The encodings in this description are named to match the encodings of CSINC.
- The description of CSINC gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 0 0 1 1 0 1 0 0 1 1 1 1 1 != 111x | 0 1 1 1 1 1 1 | Rd
   op   Rm   cond   o2   Rn

32-bit (sf == 0)

CSET <Wd>, <cond>

is equivalent to

CSINC <Wd>, WZR, WZR, invert(<cond>)

and is always the preferred disassembly.

64-bit (sf == 1)

CSET <Xd>, <cond>

is equivalent to

CSINC <Xd>, XZR, XZR, invert(<cond>)

and is always the preferred disassembly.

Assembler Symbols

<Wd>           Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>           Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<cond>         Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSETM

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This is an alias of CSINV. This means:

- The encodings in this description are named to match the encodings of CSINV.
- The description of CSINV gives the operational pseudocode for this instruction.

32-bit (sf == 0)

CSETM <Wd>, <cond>

is equivalent to

CSINV <Wd>, WZR, WZR, invert(<cond>)

and is always the preferred disassembly.

64-bit (sf == 1)

CSETM <Xd>, <cond>

is equivalent to

CSINV <Xd>, XZR, XZR, invert(<cond>)

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<cond> Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

The description of CSINV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSINC

Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is used by the aliases CINC, and CSET.

32-bit (sf == 0)

CSINC <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSINC <Xd>, <Xn>, <Xm>, <cond>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINC</td>
<td>Rm != '11111' &amp; &amp; cond != '111x' &amp; &amp; Rn != '11111' &amp; &amp; Rn == Rm</td>
</tr>
<tr>
<td>CSET</td>
<td>Rm == '11111' &amp; &amp; cond != '111x' &amp; &amp; Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;
bits(datasize) operand1 = \(X[n]\);
bits(datasize) operand2 = \(X[m]\);

if ConditionHolds(cond) then
    result = operand1;
else
    result = operand2 + 1;

\(X[d]\) = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
CSINV

Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the bitwise inversion value of the second source register.

This instruction is used by the aliases CINV, and CSETM.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>cond</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CSINV <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSINV <Xd>, <Xn>, <Xm>, <cond>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Alias Conditions

Alias | Is preferred when
---|---
CINV | Rm != '1111' && cond != '111x' && Rn != '1111' && Rn == Rm
CSETM | Rm == '11111' && cond != '111x' && Rn == '11111'

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    result = operand1;
else
    result = NOT(operand2);
X[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CSNEG

Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the negated value of the second source register.

This instruction is used by the alias **CNEG**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>cond</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>02</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

CSNEG <Wd>, <Wn>, <Wm>, <cond>

64-bit (sf == 1)

CSNEG <Xd>, <Xn>, <Xm>, <cond>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<cond> Is one of the standard conditions, encoded in the "cond" field in the standard way.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNEG</td>
<td>cond != '111x' &amp; Rn == Rm</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];

if ConditionHolds(cond) then
    result = operand1;
else
    result = NOT(operand2);
result = result + 1;
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Data Cache operation. For more information, see \texttt{op0==0b01, cache maintenance, TLB maintenance, and address translation instructions}.

This is an alias of \texttt{SYS}. This means:

- The encodings in this description are named to match the encodings of \texttt{SYS}.
- The description of \texttt{SYS} gives the operational pseudocode for this instruction.

<dc_op> Is a DC instruction name, as listed for the DC system instruction group, encoded in "op1:CRm:op2":

\begin{center}
\begin{tabular}{ccccccc}
\textbf{op1} & \textbf{CRm} & \textbf{op2} & \textbf{<dc_op>} & \textbf{Architectural Feature} \\
000 & 0110 & 001 & IVAC & - \\
000 & 0110 & 010 & ISW & - \\
000 & 0110 & 011 & IGVAC & FEAT\_MTE \\
000 & 0110 & 100 & IGSW & FEAT\_MTE \\
000 & 0110 & 101 & IGDVAC & FEAT\_MTE \\
000 & 0110 & 110 & IGDSW & FEAT\_MTE \\
000 & 1010 & 010 & CSw & - \\
000 & 1010 & 100 & CGSw & FEAT\_MTE \\
000 & 1010 & 110 & CGDSW & FEAT\_MTE \\
000 & 1110 & 010 & CISw & - \\
000 & 1110 & 100 & CIGSw & FEAT\_MTE \\
000 & 1110 & 110 & CIGDSW & FEAT\_MTE \\
011 & 0100 & 001 & ZVA & - \\
011 & 0100 & 011 & GVA & FEAT\_MTE \\
011 & 0100 & 100 & GZVA & FEAT\_MTE \\
011 & 1010 & 001 & CVAC & - \\
011 & 1010 & 011 & CGVAC & FEAT\_MTE \\
011 & 1010 & 101 & CGDVAC & FEAT\_MTE \\
011 & 1011 & 001 & CVAU & - \\
011 & 1100 & 001 & CVAP & FEAT\_DPB \\
011 & 1100 & 011 & CVGAP & FEAT\_MTE \\
011 & 1100 & 101 & CGDVAP & FEAT\_MTE \\
011 & 1101 & 001 & CVADP & FEAT\_DPB2 \\
011 & 1101 & 011 & CVGADP & FEAT\_MTE \\
011 & 1101 & 101 & CGDVADP & FEAT\_MTE \\
011 & 1110 & 001 & CIVAC & - \\
011 & 1110 & 011 & CIGVAC & FEAT\_MTE \\
011 & 1110 & 101 & CIGDVAC & FEAT\_MTE \\
\end{tabular}
\end{center}

\texttt{<op1>} Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

\texttt{<Cm>} Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

\texttt{<op2>} Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

\texttt{<Xt>} Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

The description of \texttt{SYS} gives the operational pseudocode for this instruction.
DCPS1

Debug Change PE State to EL1, when executed in Debug state:
- If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:
- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:
- \( ELR_{ELx} \) becomes UNKNOWN.
- \( SPSR_{ELx} \) becomes UNKNOWN.
- \( ESR_{ELx} \) becomes UNKNOWN.
- \( DLR_{EL0} \) and \( DSPSR_{EL0} \) become UNKNOWN.
- The endianness is set according to \( SCTLR_{ELx} \) EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and \( HCR_{EL2}.TGE == 1 \).

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see DCPS.

**Assembler Symbols**

\(<\text{imm}>\)

Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

**Operation**

\[
\text{DCPSInstruction}(\text{LL});
\]
DCPS2

Debug Change PE State to EL2, when executed in Debug state:

• If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
• Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

• EL2 if the instruction is executed at an exception level that is not EL3.
• EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

• ELR_ELx becomes UNKNOWN.
• SPSR_ELx becomes UNKNOWN.
• ESR_ELx becomes UNKNOWN.
• DLR_EL0 and DSPSR_EL0 become UNKNOWN.
• The endianness is set according to SCTLR_ELx EE.

This instruction is UNDEFINED at the following exception levels:

• All exception levels if EL2 is not implemented.
• At EL0 and EL1 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see DCPS.

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

DCPSInstruction(LL);

if !Halted() then UNDEFINED;
DCPS3

Debug Change PE State to EL3, when executed in Debug state:

• If executed at EL3 selects SP_EL3.
• Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

• ELR_EL3 becomes UNKNOWN.
• SPSR_EL3 becomes UNKNOWN.
• ESR_EL3 becomes UNKNOWN.
• DLR_EL0 and DSPSR_EL0 become UNKNOWN.
• The endianness is set according to SCTLR_EL3.EE.

This instruction is UNDEFINED at all exception levels if either:

• EDSCR.SDD == 1.
• EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPSn instructions, see DCPS.

Assembler Symbols

<imm> Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the
"imm16" field.

Operation

DCPSInstruction(LL);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DGH

DGH is a hint instruction. A DGH instruction is not expected to be performance optimal to merge memory accesses with Normal Non-cacheable or Device-GRE attributes appearing in program order before the hint instruction with any memory accesses appearing after the hint instruction into a single memory transaction on an interconnect.

System
(Armv8.6)

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

DGH

if !HaveDGHExt() then EndOfInstruction();

Operation

Hint_DGH();
DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier.

\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

DMB <option>\mid<imm>

```markdown
<table>
<thead>
<tr>
<th>CRm&lt;3:2&gt;</th>
<th>MBReqDomain_OuterShareable</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRm&lt;1:0&gt;</td>
<td>MBReqDomain_FullSystem</td>
</tr>
</tbody>
</table>
```

case CRm<3:2> of
  when '00' domain = MBReqDomain_OuterShareable;
  when '01' domain = MBReqDomain_Nonshareable;
  when '10' domain = MBReqDomain_InnerShareable;
  when '11' domain = MBReqDomain_FullSystem;

```markdown
<table>
<thead>
<tr>
<th>CRm&lt;1:0&gt;</th>
<th>MBReqTypes_All; domain = MBReqDomain_FullSystem;</th>
</tr>
</thead>
</table>
``` case CRm<1:0> of
  when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypes_Writes;
  when '11' types = MBReqTypes_All;

Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.
OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

<imm>
Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field.

Operation

DataMemoryBarrier(domain, types);
DRPS

Debug restore process state.

```
1 1 0 1 0 1 1 0 1 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
```

if !Halted() || PSTATE.EL == EL0 then UNDEFINED;

Operation

DRPSInstruction();
DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier.

It has encodings from 2 classes: Memory barrier and Memory nXS barrier

Memory barrier

<table>
<thead>
<tr>
<th>CRm</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x00</td>
</tr>
</tbody>
</table>

DSB <option>|#<imm>

case CRm<3:2> of
  when '00' domain = MBRegDomain_OuterShareable;
  when '01' domain = MBRegDomain_Nonshareable;
  when '10' domain = MBRegDomain_InnerShareable;
  when '11' domain = MBRegDomain_FullSystem;

case CRm<1:0> of
  when '00' types = MBRegTypes_All; domain = MBRegDomain_FullSystem;
  when '01' types = MBRegTypes_Reads;
  when '10' types = MBRegTypes_Writes;
  when '11' types = MBRegTypes_All;

Memory nXS barrier

(Armv8.7)

<table>
<thead>
<tr>
<th>CRm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

DSB <option>nXS|#<imm>

Assembler Symbols

<option> For the memory barrier variant: specifies the limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.
ISHLD

Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH

Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST

Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD

Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST

Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD

Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm, other than the values 0b0000 and 0b0100, that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see Data Memory Barrier (DMB) or see Data Synchronization Barrier (DSB).

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

For the memory nXS barrier variant: specifies the limitation on the barrier operation. Values are:

SY

Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm<3:2> = 0b11.

ISH

Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b10.

NSH

Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b01.

OSH

Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm<3:2> = 0b00.

<imm>

For the memory barrier variant: is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

For the memory nXS barrier variant: is a 5-bit unsigned immediate, encoded in “CRm<3:2>”:

<table>
<thead>
<tr>
<th>CRm&lt;3:2&gt;</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>16</td>
</tr>
<tr>
<td>01</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>28</td>
</tr>
</tbody>
</table>
Operation

if HaveTME() && TSTATE.depth > 0 then
  FailTransaction(TMFailure_ERR, FALSE);
  DataSynchronizationBarrier(domain, types);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DVP

Data Value Prediction Restriction by Context prevents data value predictions, based on information gathered from earlier execution within an particular execution context, from allowing later speculative execution within that context to be observable through side-channels.

For more information, see DVP RCTX, Data Value Prediction Restriction by Context.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

System
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | Rt |

DVP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #5, <Xt>

and is always the preferred disassembly.

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.

\[
\begin{array}{cccccccccccccccc}
\hline
sf & 1 & 0 & 0 & 1 & 0 & 1 & 0 & | & shift & 1 & | & Rm & | & imm6 & | & Rn & | & Rd & \\
opc & N
\end{array}
\]

32-bit (sf == 0)

EON \(<Wd>, \,<Wn>, \,<Wm>{, \,<\text{shift} \, #<\text{amount}>}\)

64-bit (sf == 1)

EON \(<Xd>, \,<Xn>, \,<Xm>{, \,<\text{shift} \, #<\text{amount}>}\)

integer \(d\) = \(\text{UInt}(Rd)\);
integer \(n\) = \(\text{UInt}(Rn)\);
integer \(m\) = \(\text{UInt}(Rm)\);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

\begin{itemize}
  \item \textbf{ShiftType} \(\text{shift}\)\_\text{type} = \text{DecodeShift}(\text{shift});
  \item integer \(\text{shift}\)\_\text{amount} = \text{UInt}(\text{imm6});
\end{itemize}

Assembler Symbols

\begin{itemize}
  \item \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
  \item \(<Wn>\) Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
  \item \(<Wm>\) Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
  \item \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
  \item \(<Xn>\) Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
  \item \(<Xm>\) Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
  \item \(<\text{shift}>\) Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":
\end{itemize}

\[
\begin{array}{c|c}
\text{shift} & \langle\text{shift}\rangle \\
\hline
00 & LSL \\
01 & LSR \\
10 & ASR \\
11 & ROR \\
\end{array}
\]

\begin{itemize}
  \item \(<\text{amount}>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  \item For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
\end{itemize}

Operation

\[
\begin{align*}
\text{bits(datasize)} \, \text{operand1} &= \bar{x}[n]; \\
\text{bits(datasize)} \, \text{operand2} &= \text{ShiftReg}(m, \text{shift}\_\text{type}, \text{shift}\_\text{amount}); \\
\text{operand2} &= \text{NOT}(\text{operand2}); \\
\text{result} &= \text{operand1} \text{ EOR } \text{operand2}; \\
x[d] &= \text{result};
\end{align*}
\]
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**EOR (immediate)**

Bitwise Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit \((sf == 0 && N == 0)\)

EOR \(<Wd|WSP>, <Wn>, #<imm>\)

### 64-bit \((sf == 1)\)

EOR \(<Xd|SP>, <Xn>, #<imm>\)

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);
```

**Assembler Symbols**

- `<Wd|WSP>`  Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn>`  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd|SP>`  Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn>`  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<imm>`  For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
result = operand1 EOR imm;
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
EOR (shifted register)

Bitwise Exclusive OR (shifted register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

EOR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

EOR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

\[
\text{bits(datasize) operand1} = X[n];
\text{bits(datasize) operand2} = \text{ShiftReg}(m, shift\_type, shift\_amount);
\text{result} = \text{operand1 EOR operand2};
\text{X[d]} = \text{result};
\]

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores \texttt{PSTATE} from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See \textit{Illegal return events from AArch64 state}.

\texttt{ERET} is \texttt{UNDEFINED} at \texttt{EL0}.

\begin{verbatim}
1 1 0 1 0 1 1 0 1 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0
A M Rn op4
\end{verbatim}

**Operation**

\begin{verbatim}
\texttt{AArch64.CheckForERetTrap(FALSE, TRUE);}  
\texttt{bits(64) target = ELR[];}  
\texttt{AArch64.ExceptionReturn(target, SPSR[]);}
\end{verbatim}
ERETAA, ERETABB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores PSTATE from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA, and key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See Illegal return events from AArch64 state.

ERETAA and ERETAB are UNDEFINED at EL0.

Integer
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 01 | 01 | 01 | 00 | 01 | 01 | 01 | 11 | 11 | 11 | 00 | 00 | 00 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |

ERETAA (M == 0)

ERETAA

ERETAB (M == 1)

ERETAB

if PSTATE.EL == EL0 then UNDEFINED;

boolean use_key_a = (M == '0');

if !HavePACExt() then

UNDEFINED;

Operation

AArch64.CheckForERetTrap(TRUE, use_key_a);

bits(64) target;

if use_key_a then

target = AuthIA(ELR[], SP[], TRUE);
else

target = AuthIB(ELR[], SP[], TRUE);

AArch64.ExceptionReturn(target, SPSR[]);
Error Synchronization Barrier is an error synchronization event that might also update DISR_EL1 and VDISR_EL2. This instruction can be used at all Exception levels and in Debug state. In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the Arm(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile. If the RAS Extension is not implemented, this instruction executes as a NOP.

System
(Armv8.2)

```
if !HaveRASExt() then EndOfInstruction();
```

Operation

```
if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);
    SynchronizeErrors();
    AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
```
**EXTR**

Extract register extracts a register from a pair of registers.

This instruction is used by the alias **ROR (immediate)**.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|---------------|---------------|---------------|---------------|
| sf | 0 | 0 | 1 | 0 | 0 | 1 | 1 | N | 0 | Rm | imms | Rn | Rd |

**32-bit (sf == 0 && N == 0 && imms == 0xxxxx)**

EXTR `<Wd>`, `<Wn>`, `<Wm>`, `#<lsb>`

**64-bit (sf == 1 && N == 1)**

EXTR `<Xd>`, `<Xn>`, `<Xm>`, `#<lsb>`

```plaintext
def d = UInt(Rd);
def n = UInt(Rn);
def m = UInt(Rm);
def datasize = if sf == '1' then 64 else 32;
def lsb;

if N != sf then UNDEFINED;
if sf == '0' && imms<5> == '1' then UNDEFINED;
lsb = UInt(imms);
```

**Assemble Symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>`: Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<lsb>`: For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field.
  
  For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR (immediate)</td>
<td>Rn == Rm</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(2*datasize) concat = operand1:operand2;
result = concat<lsb+datasize-1:lsb+1>;
X[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
GMI

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

Integer
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  |

GMI <Xd>, <Xn|SP>, <Xm>

if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field.

Operation

bits(64) address = if n == 31 then SP[] else X[n];
bits(64) mask = X[m];
bits(4) tag = AArch64.AllocationTagFromAddress(address);
mask<UInt(tag)> = '1';
X[d] = mask;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Hint instruction is for the instruction set space that is reserved for architectural hint instructions. Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 1 0 0 0 1 0 0 1 0 | CRm | op2 | 1 1 1 1 1

SystemHintOp op;
case CRm:op2 of
  when '0000 000' op = SystemHintOp_NOP;
  when '0000 001' op = SystemHintOp_YIELD;
  when '0000 010' op = SystemHintOp_WFE;
  when '0000 011' op = SystemHintOp_WFI;
  when '0000 100' op = SystemHintOp_SEV;
  when '0000 101' op = SystemHintOp_SEVL;
  when '0000 110'
    if !HaveDGHExt() then EndOfInstruction(); // Instruction executes as NOP
    op = SystemHintOp_DGH;
  when '0000 111' SEE "XPACLRI";
  when '0001 xxx'
    case op2 of
      when '000' SEE "PACIA1716";
      when '010' SEE "PACIB1716";
      when '100' SEE "AUTIA1716";
      when '110' SEE "AUTIB1716";
      otherwise EndOfInstruction();
    when '0010 000'
      if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
      op = SystemHintOp_ESB;
    when '0010 001'
      if !HaveStatisticalProfiling() then EndOfInstruction(); // Instruction executes as NOP
      op = SystemHintOp_PSB;
    when '0010 010'
      if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
      op = SystemHintOp_TSB;
    when '0010 100'
      op = SystemHintOp_CSDB;
    when '0011 xxx'
      case op2 of
        when '000' SEE "PACIAZ";
        when '001' SEE "PACIASP";
        when '010' SEE "PACIBZ";
        when '011' SEE "PACIBSP";
        when '100' SEE "AUTIAZ";
        when '101' SEE "AUTHASP";
        when '110' SEE "AUTIBZ";
        when '111' SEE "AUTIBSP";
        when '0100 xx0'
          op = SystemHintOp_BTI;
          // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
          SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
        otherwise EndOfInstruction();

Assembler Symbols

<imm> Is a 7-bit unsigned immediate, in the range 0 to 127 encoded in the "CRm:op2" field.
The encodings that are allocated to architectural hint functionality are described in the "Hints" table in
the "Index by Encoding".
For allocated encodings of "CRm:op2":

HINT
• A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
• An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.

**Operation**

```
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_DGH
    Hint_DGH();
  when SystemHintOp_WFE
    Hint_WFE(-1);
  when SystemHintOp_WFI
    Hint_WFI(-1);
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    if HaveTME() && TSTATE.depth > 0 then
      FailTransaction(TMFailure_ERR, FALSE);
      SynchronizeErrors();
      AArch64.ESBOperation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
      AArch64.vESBOperation();
      TakeUnmaskedSErrorInterrupts();
  when SystemHintOp_PSB
    ProfilingSynchronizationBarrier();
  when SystemHintOp_TSB
    TraceSynchronizationBarrier();
  when SystemHintOp_CSDB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    SetBTypeNext('00');
  otherwise    // do nothing
```
**HLT**

Halt instruction. An HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

```
1 1 0 1 0 1 0 0 0 1 0 | imm16 | 0 0 0 0 0
```

HLT #<imm>

```javascript
if ESCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if HaveBTIExt() then
  SetBTypeCompatible(TRUE);
```

**Assembler Symbols**

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

**Operation**

```
Halt(DebugHalt_HaltInstruction);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
HVC

Hypervisor Call causes an exception to EL2. Non-secure software executing at EL1 can use this instruction to call the hypervisor to request a service.

The HVC instruction is UNDEFINED:

- At EL0.
- At EL1 if EL2 is not enabled in the current Security state.
- When SCR_EL3.HCE is set to 0.

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in ESR_ELx, using the EC value 0x16, and the value of the immediate argument.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0</td>
</tr>
</tbody>
</table>

HVC #<imm>

// Empty.

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

\[
\text{if } \neg \text{HaveEL(EL2)} \lor \text{PSTATE.EL} = \text{EL0} \lor \text{PSTATE.EL} = \text{EL1} \land \neg \text{IsSecureEL2Enabled()} \land \text{IsSecure()} \text{ then UNDEFINED;}
\]

\[
hvc\_enable = \text{if HaveEL(EL3) then SCR\_EL3.HCE else NOT(HCR\_EL2.HCD)};
\]

\[
\text{if hvc\_enable == '0' then UNDEFINED; else AArch64.CallHypervisor(imm16);}
\]
IC

Instruction Cache operation. For more information, see \( op0==0b01, cache \) maintenance, \( TLB \) maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | CRm| op2| Rt |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
    |<ic_op>|,<Xt>|
```

IC \(<ic\_op>, <Xt>\) is equivalent to

\( SYS \#<op1>, C7, <Cm>, \#<op2>, <Xt>\)

and is the preferred disassembly when \( SysOp(op1,'0111',CRm,op2) == Sys_IC \).

Assembler Symbols

\(<ic\_op>\) is an IC instruction name, as listed for the IC system instruction pages, encoded in "op1:CRm:op2":

<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>&lt;ic_op&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0001</td>
<td>000</td>
<td>IALLUIS</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>000</td>
<td>IALLU</td>
</tr>
<tr>
<td>011</td>
<td>0101</td>
<td>001</td>
<td>IVAU</td>
</tr>
</tbody>
</table>

\(<op1>\) is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

\(<Cm>\) is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

\(<op2>\) is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

\(<Xt>\) is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

The description of SYS gives the operational pseudocode for this instruction.
IRG

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

Integer
(Armv8.5)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xm</td>
</tr>
</tbody>
</table>

IRG <Xd|SP>, <Xn|SP>>{, <Xm}>

if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

Assembler Symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

Operation

bits(64) operand = if n == 31 then SP[] else X[n];
bits(64) exclude_reg = X[m];
bits(16) exclude = exclude_reg<<15:0> OR GCR_EL1.Exclude;

if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    if GCR_EL1.RRND == '1' then
        RGSR_EL1 = bits(64) UNKNOWN;
        if IsOnes(exclude) then
            rtag = '0000';
        else
            rtag = ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start = RGSR_EL1.TAG;
        bits(4) offset = AArch64.RandomTag();
        rtag = AArch64.ChooeNonExcludedTag(start, offset, exclude);
    else
        rtag = '0000';

bits(64) result = AArch64.AddressWithAllocationTag(operand, AccType_NORMAL, rtag);

if d == 31 then
    SP[] = result;
else
    X[d] = result;
ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see Instruction Synchronization Barrier (ISB).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 1 0 1 1 0 1 1 0 1 1 1 1
```

```
ISB {<option>|#<imm>}
```

// No additional decoding required

**Assembler Symbols**

- `<option>` Specifies an optional limitation on the barrier operation. Values are:
  - **SY**
    
    Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

    All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

- `<imm>` Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

**Operation**

```
InstructionSynchronizationBarrier();
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD64B

Single-copy Atomic 64-byte Load derives an address from a base register value, loads eight 64-bit doublewords from a memory location, and writes them to consecutive registers, Xt to X(t+7). The data that is loaded is atomic and is required to be 64-byte aligned.

Integer
(Armv8.7)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD64B <Xt>, [<Xn|SP> {,#0}]

if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemLoad64B(address, acctype);

for i = 0 to 7
    value = data<63+64*i:64*i>;
    if BigEndian(acctype) then value = BigEndianReverse(value);
    X[t+i] = value;
LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDADDA and LDADDAL load from memory with acquire semantics.
- LDADDL and LDADDAL store to memory with release semantics.
- LDADD has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STADD, STADDL.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs  | 0  | 0  | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>
32-bit LDADD (size == 10 && A == 0 && R == 0)

LDADD <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDA (size == 10 && A == 1 && R == 0)

LDADDA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDAL (size == 10 && A == 1 && R == 1)

LDADDAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDADDL (size == 10 && A == 0 && R == 1)

LDADDL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDADD (size == 11 && A == 0 && R == 0)

LDADD <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDA (size == 11 && A == 1 && R == 0)

LDADDA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDAL (size == 11 && A == 1 && R == 1)

LDADDAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDADDL (size == 11 && A == 0 && R == 1)

LDADDL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDERED_ATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADD, STADDL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDADDAB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDB and LDADDALB store to memory with release semantics.
- LDADDB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STADDB, STADDLB.

Integer
(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDADDAB (A == 1 && R == 0)

LDADDAB <Ws>, <Wt>, [<Xn|SP>]

LDADDALB (A == 1 && R == 1)

LDADDALB <Ws>, <Wt>, [<Xn|SP>]

LDADDB (A == 0 && R == 0)

LDADDB <Ws>, <Wt>, [<Xn|SP>]

LDADDLB (A == 0 && R == 1)

LDADDLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

t = UInt(Rt);
n = UInt(Rn);
s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDB, STADDLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDADDH, LDADDAH, LDADDALH, LDADDLH**

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDADDAH` and `LDADDALH` load from memory with acquire semantics.
- `LDADDLH` and `LDADDALH` store to memory with release semantics.
- `LDADDH` has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.* For information about memory accesses see *Load/Store addressing modes.*

This instruction is used by the alias `STADDH, STADDLH`.

**Integer**
(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**size**  
**opc**

**LDADDAH (A == 1 && R == 0)**

`LDADDAH <Ws>, <Wt>, [<Xn|SP>]`

**LDADDALH (A == 1 && R == 1)**

`LDADDALH <Ws>, <Wt>, [<Xn|SP>]`

**LDADDH (A == 0 && R == 0)**

`LDADDH <Ws>, <Wt>, [<Xn|SP>]`

**LDADDLH (A == 0 && R == 1)**

`LDADDLH <Ws>, <Wt>, [<Xn|SP>]`

if `!HaveAtomicExt()` then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

boolean tag_checked = n != 31;

**Assembler Symbols**

- `<Ws>`  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Wt>`  Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>`  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STADDH, STADDLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

\[
\begin{align*}
\text{bits}(64) & \quad \text{address;} \\
\text{bits}(16) & \quad \text{value;} \\
\text{bits}(16) & \quad \text{data;} \\
\end{align*}
\]

if \( \text{HaveMTEExt}() \) then
  \( \text{SetTagCheckedInstruction}(\text{tag}\_\text{checked}) \);

\[\text{value} = \text{X}[s];\]
if \( n = 31 \) then
  \( \text{CheckSPAlignment}(); \)
  \( \text{address} = \text{SP}[]; \)
else
  \( \text{address} = \text{X}[n]; \)
\[\text{data} = \text{MemAtomic}(\text{address}, \text{MemAtomicOp}\_\text{ADD}, \text{value}, \text{ldacctype}, \text{stacctype});\]
if \( t \neq 31 \) then
  \( \text{X}[t] = \text{ZeroExtend}(\text{data}, 32); \)

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

---

### Integer

(Armv8.3)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|   1 |   1 |   1 |   0 |   0 |   1 |   0 |   1 |   1 |   1 |   1 |   1 |   1 |   1 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |   0 |

**32-bit (size == 10)**

```plaintext
LDAPR <Wt>, [<Xn|SP> {,#0}]
```

**64-bit (size == 11)**

```plaintext
LDAPR <Xt>, [<Xn|SP> {,#0}]
```

### Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.
- `<Xt>`: Is the 64-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

---

**Integer**  
*(Armv8.3)*

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>Rs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LDAPRB**  
<Wt>, [<Xn|SP> {,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

**Assembler Symbols**

- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```plaintext
bits(64) address;
bits(8) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.
For information about memory accesses, see *Load/Store addressing modes*.

### Integer

**(Armv8.3)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>Rt</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

** LDAPRH  
<Wt>, [<Xn|SP> {,#0}]**

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the “Rt” field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a
32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*,
except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release,
  created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does
  not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

### Unscaled offset

(ARMv8.4)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------|----------------|------------------|
| size                                | opc            | imm9             | 0 0 | Rn | Rt |

#### 32-bit (size == 10)

LDAPUR `<Wt>`, `[<Xn|SP>{, #<simm>}]`

#### 64-bit (size == 11)

LDAPUR `<Xt>`, `[<Xn|SP>{, #<simm>}]`

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in
  the "imm9" field.

### Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
Operation

if `HaveMTEExt()` then
    `SetTagCheckedInstruction(tag_checked)``;

bits(64) address;
bits(datasize) data;

if n == 31 then
    `CheckSPA1ignment()``;
    address = `SP[``];
else
    address = `X[n]``;

address = address + offset;

data = `Mem[address, datasize DIV 8, AccType ORDERED]``;
`X[t] = ZeroExtend(data, regsize)``;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURB

Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

### Unscaled offset

*(Armv8.4)*

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | Rn | Rt |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----+----|

LDAPURB <Wt>, [<Xn|SP>{, #<simm>}

bits(64) offset = SignExtend(imm9, 64);
```

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```java
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

### Operation

```java
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

### Unscaled offset
*(Armv8.4)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Unscaled offset**

```
bits(64) offset = SignExtend(imm9, 64);
```

#### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

#### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

#### Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

#### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDAPURH
LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(ARMv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 1  | 1  | x  | 0  | imm9| 0  | 0  | Rn | Rrt|

size opc

32-bit (opc == 11)

LDAPURSB <Wt>, [{<Xn|SP>{, #<simm>}}]

64-bit (opc == 10)

LDAPURSB <Xt>, [{<Xn|SP>{, #<simm>}}]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 1, AccType_ORDERED] = data;
    when MemOp_LOAD
        data = Mem[address, 1, AccType_ORDERED];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURSH

Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see Load/Store addressing modes.

Unscaled offset

(ARMv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 1  | x  | 0  | imm9| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**32-bit (opc == 11)**

LDAPURSH <Wt>, [/<Xn|SP>{, #<simm>}

**64-bit (opc == 10)**

LDAPURSH <Xt>, [/<Xn|SP>{, #<simm>}

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

-Wt- Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
-Xt- Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
-Xn|SP>- Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
-simm>- Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bites(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_ORDERED] = data;
    when MemOp_LOAD
        data = Mem[address, 2, AccType_ORDERED];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

**Unscaled offset**

(Armv8.4)

|     | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|size| 0  | 0  | 1  | 0  | 1  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|     | imm9|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|opc | Rn  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

LDAPURSW `<Xt>, [<Xn|SP>{, #<simm>}]`

bits(64) offset = `SignExtend`(imm9, 64);

**Assembler Symbols**

- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

integer n = `UInt`(Rn);
integer t = `UInt`(Rt);

boolean tag_checked = n != 31;

**Operation**

if `HaveMTEExt()` then
    `SetTagCheckedInstruction`(tag_checked);

bits(64) address;
bits(32) data;

if n == 31 then
    `CheckSPAlignment`();
    address = `SP`[];
else
    address = `X`[n];

address = address + offset;

data = `Mem`[address, 4, `AccType_ORDERED`];
`X`[t] = `SignExtend`(data, 64);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

32-bit (size == 10)

LDAR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDAR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
else
    address = X[n];

data = Mem[address, dbytes, AccType_ORDERED];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 1 1 0 [(1)(1)(1)(1)(1)(1) 1 [(1)(1)(1)(1)(1)(1)] Rn Rt

LDARB <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 1, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 1 0 0 1 1 0 |(1)(1)(1)(1)(1)(1)(1)(1)| 1 |(1)(1)(1)(1)(1)|
size          L      Rs    o0  Rt2
```

LDARH <Wt>, [<Xn|SP>{,#0}]

- `integer n = UInt(Rn);`
- `integer t = UInt(Rt);`
- `boolean tag_checked = n != 31;`

Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = Mem[address, 2, AccType_ORDERED];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

32-bit (sz == 0)

LDAXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

LDAXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
  Constraint c = ConstraineUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDAXP.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

```plaintext
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);
if rt unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
    if BigEndian(AccType_ORDEREDATOMIC) then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_ORDEREDATOMIC, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ORDEREDATOMIC];
    X[t2] = Mem[address+8, 8, AccType_ORDEREDATOMIC];
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit
doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address
being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See
Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire,
Store-Release. For information about memory accesses see Load/Store addressing modes.

### 32-bit (size == 10)

LDAXR <Wt>, [<Xn|SP>{,#0}]

### 64-bit (size == 11)

LDAXR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

#### Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

#### Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
SetTagCheckedInstruction(tag_checked);
if n == 31 then
CheckSPAlignment();
   address = SP[0];
else
   address = X[n];
// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);
data = Mem[address, dbytes, AccType_ORDERED_ATOMIC];
X[t] = ZeroExtend(data, regsize);

#### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
size L Rs o0 Rt2

LDAXRB <Wt>, [<Xn|SP>{, #0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ORDEREDATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  |

LDAXRH <Wt>, [<Xn|SP>{,=#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ORDEREDATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LDCLR, LDCLRA, LDCLRAL, LDCLRL**

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDCLRA and LDCLRAL load from memory with acquire semantics.
- LDCLRL and LDCLRAL store to memory with release semantics.
- LDCLR has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STCLR, STCLRL**.

### Integer

*(Armv8.1)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 0  | 1  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**size**

**opc**
32-bit LDCLR (size == 10 && A == 0 && R == 0)
LDCLR <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRA (size == 10 && A == 1 && R == 0)
LDCLRA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRAL (size == 10 && A == 1 && R == 1)
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDCLRL (size == 10 && A == 0 && R == 1)
LDCLRL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDCLR (size == 11 && A == 0 && R == 0)
LDCLR <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRA (size == 11 && A == 1 && R == 0)
LDCLRA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRAL (size == 11 && A == 1 && R == 1)
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDCLRL (size == 11 && A == 0 && R == 1)
LDCLRL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLR, STCLRL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
**Operation**

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if `HaveMTEExt()` then:
    `SetTagCheckedInstruction(tag_checked);`

value = `X[s];`
if n == 31 then
    `CheckSPAlignment();`
    address = `SP[];`
else
    address = `X[n];`

data = `MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);`

if t != 31 then
    `X[t] = ZeroExtend(data, regsize);`

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDCLR, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRLB and LDCLRALB store to memory with release semantics.
- LDCLR has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STCLR, STCLRAB.

Integer
(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LDCLR (A == 1 && R == 0)**

LDCLR <Ws>, <Wt>, [<Xn|SP>]

**LDCLRALB (A == 1 && R == 1)**

LDCLRALB <Ws>, <Wt>, [<Xn|SP>]

**LDCLRLB (A == 0 && R == 1)**

LDCLRLB <Ws>, <Wt>, [<Xn|SP>]

**LDCLR (A == 0 && R == 0)**

LDCLR <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLR, STCLRLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- LDCLRH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STCLRH, STCLRLH.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A | R | 1 | Rs | 0 | 0 | 0 | 1 | 0 | 0 | Rn | Rt |

size opc

**LDCLRAH (A == 1 && R == 0)**

LDCLRAH <Ws>, <Wt>, [Xn|SP]

**LDCLRALH (A == 1 && R == 1)**

LDCLRALH <Ws>, <Wt>, [Xn|SP]

**LDCLRH (A == 0 && R == 0)**

LDCLRH <Ws>, <Wt>, [Xn|SP]

**LDCLRLH (A == 0 && R == 1)**

LDCLRLH <Ws>, <Wt>, [Xn|SP]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STCLRH, STCLRLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDEORA and LDEORAL load from memory with acquire semantics.
- LDEORL and LDEORAL store to memory with release semantics.
- LDEOR has neither acquire nor release semantics.

For more information about memory ordering semantics see `Load-Acquire, Store-Release`.
For information about memory accesses see `Load/Store addressing modes`.

This instruction is used by the alias STEOR, STEORL.

**Integer**

*(Armv8.1)*

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1   | x   | 1   | 1   | 1   | 0   | 0   | 0   | A   | R   | 1   | Rs  | 0   | 0   | 1   | 0   | 0   | Rn  | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |

size  opc
32-bit LDEOR (size == 10 & A == 0 & R == 0)
LDEOR <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORA (size == 10 & A == 1 & R == 0)
LDEORA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORAL (size == 10 & A == 1 & R == 1)
LDEORAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDEORL (size == 10 & A == 0 & R == 1)
LDEORL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDEOR (size == 11 & A == 0 & R == 0)
LDEOR <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORA (size == 11 & A == 1 & R == 0)
LDEORA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORAL (size == 11 & A == 1 & R == 1)
LDEORAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDEORL (size == 11 & A == 0 & R == 1)
LDEORL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEOR, STEORL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with
the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is
returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STEORB, STEORLB.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

size opc

LDEORAB (A == 1 && R == 0)
LDEORAB <Ws>, <Wt>, [<Xn|SP>]

LDEORALB (A == 1 && R == 1)
LDEORALB <Ws>, <Wt>, [<Xn|SP>]

LDEORB (A == 0 && R == 0)
LDEORB <Ws>, <Wt>, [<Xn|SP>]

LDEORLB (A == 0 && R == 1)
LDEORLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

Ws Is the 32-bit name of the general-purpose register holding the data value to be operated on with the
contents of the memory location, encoded in the "Rs" field.
Wt Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
Xn|SP Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORB, STEORLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STEORH, STEORLH.

### Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | Rn | 1  | Rt |

size opc

#### LDEORAH (A == 1 && R == 0)

LDEORAH <Ws>, <Wt>, [<Xn|SP>]

#### LDEORALH (A == 1 && R == 1)

LDEORALH <Ws>, <Wt>, [<Xn|SP>]

#### LDEORH (A == 0 && R == 0)

LDEORH <Ws>, <Wt>, [<Xn|SP>]

#### LDEORLH (A == 0 && R == 1)

LDEORLH <Ws>, <Wt>, [<Xn|SP>]

```plaintext
if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

### Assembler Symbols

- <Ws> is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORH, STEORLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);
if t != 31 then
  X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDG

Load Allocation Tag loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag granule.

Integer
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | imm9 | 0 | 0 | Xn | Xt |

LDG <Xt>, [<Xn|SP>{, #<simm>}

if !HaveMTEExt() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

bits(64) address;
bits(4) tag;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
address = Align(address, TAG_GRANULE);
tag = AArch64_MemTag[address, AccType_NORMAL];
X[t] = AArch64_AddressWithAllocationTag(X[t], AccType_NORMAL, tag);
LDGM

Load Tag Multiple reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and writes the Allocation Tag read from address A to the destination register at 4*A<7:4>+3:4*A<7:4>. Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

If ID_AA64PFR1_EL1.MTE != 0b0010, this instruction is UNDEFINED.

### Integer

(armv8.5)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 Xn</td>
</tr>
</tbody>
</table>

LDGM <Xt>, [<Xn|SP>]

if ![HaveMTE2Ext]() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

### Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

### Operation

if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = AArch64.MemTag[address, AccType_NORMAL];
    data<((index*4)+3:index*4> = tag;
    address = address + TAG_GRANULE;
    index = index + 1;

X[t] = data;
LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset
(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|----------------|-----------------|-----------------|----------------|
| 1 x 0 0 1 0 0 0 1 1 0 (1) (1) (1) (1) 0 (1) (1) (1) (1) | Rn | Rt |

size L Rs o0 Rt2

32-bit (size == 10)

LDLAR <Wt>, [<Xn|SP>]{,#0}

64-bit (size == 11)

LDLAR <Xt>, [<Xn|SP>]{,#0}

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
   SetTagCheckedInstruction(tag_checked);
if n == 31 then
   CheckSPAlignment();
   address = SP[];
else
   address = X[n];
data = Mem[address, dbytes, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

**No offset**
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | (1)(1)(1)(1)(1)| 0  | (1)(1)(1)(1)(1)| Rn  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| size | L  | Rs  | o0 | Rt2 |

LDLARB <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

**Assembler Symbols**

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

bits(64) address;
bits(8) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
data = Mem[address, 1, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDLARH**

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

### No offset
(Armv8.1)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0 | (1) | (1) | (1) | (1) | 0 | (1) | (1) | (1) | (1) |   |   |   |   |   |   |   |   |   |

<table>
<thead>
<tr>
<th>size</th>
<th>L</th>
<th>Rs</th>
<th>o0</th>
<th>Rt2</th>
</tr>
</thead>
</table>

LDLARH `<Wt>`, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

```
bits(64) address;
bits(16) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = Mem[address, 2, AccType_LIMITEDORDERED];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see Load/Store addressing modes. For information about Non-temporal pair instructions, see Load/Store Non-temporal pair.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
x 0 1 0 1 0 0 0 1 | imm7 | Rt2 | Rn | Rt
opc L

32-bit (opc == 00)

LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}

64-bit (opc == 10)

LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}

// Empty.

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDNP.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bias(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrantUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
    when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

LDNP
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
address = address + offset;
data1 = Mem[address, dbytes, AccType_STREAM];
data2 = Mem[address+dbytes, dbytes, AccType_STREAM];
if rt_unknown then
  data1 = bits(datasize) UNKNOWN;
  data2 = bits(datasize) UNKNOWN;
  X[t] = data1;
  X[t2] = data2;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**32-bit (opc == 00)**

LDP `<Wt1>`, `<Wt2>`, `[<Xn|SP>]`, `#<imm>`

**64-bit (opc == 10)**

LDP `<Xt1>`, `<Xt2>`, `[<Xn|SP>]`, `#<imm>`

boolean wback = TRUE;
boolean postindex = TRUE;

### Pre-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 1  | 0  | 1  | 0  | 0  | 1  | 1  |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**32-bit (opc == 00)**

LDP `<Wt1>`, `<Wt2>`, `[<Xn|SP>]`, `#<imm>`

**64-bit (opc == 10)**

LDP `<Xt1>`, `<Xt2>`, `[<Xn|SP>]`, `#<imm>`

boolean wback = TRUE;
boolean postindex = TRUE;

### Signed offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**32-bit (opc == 00)**

LDP `<Wt1>`, `<Wt2>`, `[<Xn|SP>]`, `#<imm>`

**64-bit (opc == 10)**

LDP `<Xt1>`, `<Xt2>`, `[<Xn|SP>]`, `#<imm>`

boolean wback = FALSE;
boolean postindex = FALSE;
For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP*.

**Assembler Symbols**

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

**Shared Decode**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bins(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if t == t2 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LDOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();
```

LDP
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
if !postindex then
  address = address + offset;

data1 = Mem[address, dbytes, AccType_NORMAL];
data2 = Mem[address+dbytes, dbytes, AccType_NORMAL];
if rt unknown then
  data1 = bits(datasize) UNKNOWN;
  data2 = bits(datasize) UNKNOWN;
if signed then
  X[t] = SignExtend(data1, 64);
  X[t2] = SignExtend(data2, 64);
else
  X[t] = data1;
  X[t2] = data2;
if wback then
  if wb unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>opc</td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>opc</td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = FALSE;

Signed offset

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>opc</td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

boolean wback = FALSE;
boolean postindex = FALSE;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDPSW.

Assembler Symbols

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bits(64) offset = LSL(SignExtend(imm7, 64), 2);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
endif

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
endif

Operation

bits(64) address;
bits(32) data1;
bits(32) data2;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
endif

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
endif

if !postindex then
    address = address + offset;
endif

data1 = Mem[address, 4, AccType_NORMAL];
data2 = Mem[address+4, 4, AccType_NORMAL];

if rt unknown then
    data1 = bits(32) UNKNOWN;
data2 = bits(32) UNKNOWN;
X[t] = SignExtend(data1, 64);
X[t2] = SignExtend(data2, 64);
endif

if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
    endif
endif

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDR (immediate)**

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

### Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 x 1 1 1 0 0 0 0 1 0 imm9 0 1 Rn Rt |

#### 32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>], #<simm>
```

#### 64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>], #<simm>
```
32-bit (size == 10)

LDR <Wt>, [<Xn|SP>{, #pimm}]

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>{, #pimm}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDR (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
simm> For the 32-bit variant: is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.
For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
case c of
    when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
    when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
    when Constraint_UNDEF undefined;
    when Constraint_NOP EndOfInstruction();
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

32-bit (opc == 00)

LDR <Wt>, <label>

64-bit (opc == 01)

LDR <Xt>, <label>

integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;
case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 4;
    signed = TRUE;
  when '11'
    memop = MemOp_PREFETCH;
offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(64) address = PC[] + offset;
bits(size*8) data;
if HaveMTEExt() then
  SetTagCheckedInstruction(FALSE);
case memop of
  when MemOp_LOAD
    data = Mem[address, size, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, 64);
    else
      X[t] = data;
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | x | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | Rm | option | S | 1 | 0 | Rn | | Rt |

size opc

32-bit (size == 10)

LDR <Wt>, [<Xn|SP>, {<Wm>|<Xm>}, {<extend> {<amount>}}]

64-bit (size == 11)

LDR < Xt >, [<Xn|SP>, {<Wm>|<Xm>}, {<extend> {<amount>}}]

integer scale = UInt(size);
if option<1> == 0 then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == 1 then scale else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>
Shared Decode

```plaintext
integer n = UINT(Rn);
integer t = UINT(Rt);
integer m = UINT(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
```
**LDRAA, LDRAB**

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for LDRAA, and key B is used for LDRAB.

If the authentication passes, the PE behaves the same as for an LDR instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see *Load/Store addressing modes*.

**Unscaled offset**

(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 0  | M  | S  | 1  | imm9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| size |

**Key A, offset (M == 0 & W == 0)**

LDRAA <Xt>, [<Xn|SP>{, #<simm>}]

**Key A, pre-indexed (M == 0 & W == 1)**

LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!

**Key B, offset (M == 1 & W == 0)**

LDRAB <Xt>, [<Xn|SP>{, #<simm>}]

**Key B, pre-indexed (M == 1 & W == 1)**

LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!

if !HavePACExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
bits(64) offset = LSL(SignExtend(S10, 64), 3);
boolean tag_checked = wback || n != 31;

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.
Operation

bits(64) address;
bits(64) data;
boolean wb_unknown = FALSE;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if n == 31 then
    address = SP[];
else
    address = X[n];

if use_key_a then
    address = AuthDA(address, X[31], TRUE);
else
    address = AuthDB(address, X[31], TRUE);

if n == 31 then
    CheckSPAlignment();

address = address + offset;
data = Mem[address, 8, AccType_NORMAL];
X[t] = data;

if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDRB (immediate)**

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

### Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | imm9| 0   | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

```
size opc

LDRB <Wt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | imm9| 1   | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

```
size opc

LDRB <Wt>, [<Xn|SP>], #<simm>!

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 1   | imm12|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

```
size opc

LDRB <Wt>, [<Xn|SP>{, #<pimm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRH (immediate)*.

**Assembler Symbols**

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>`: Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>`: Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDRB (register)**

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

Extended register (option != 011)

\[ \text{LDRB} \ <Wt>, \ [<Xn|SP>, \ (<Wm>|<Xm>), \ <extend> \ {<amount>}] \]

Shifted register (option == 011)

\[ \text{LDRB} \ <Wt>, \ [<Xn|SP>, \ <Xm>{, \ LSL <amount>}] \]

if option<1> == '0' then UNDEFINED; // sub-word index

\[ \text{ExtendType extend_type} = \text{DecodeRegExtend}(\text{option}); \]

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>` When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>` When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>` Is the index extend specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

`<amount>` Is the index shift amount, it must be #0, encoded in “S” as 0 if omitted, or as 1 if present.

**Shared Decode**

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```
Operation

bits(64) offset = ExtendReg(m, extend_type, 0);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 0 1 1 0</td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.
Shared Decode

integer n = Uint(Rn);
integer t = Uint(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
end;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
else

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>Rm</th>
<th>option</th>
<th>S</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0 0 0</td>
<td>0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDRH `<Wt>`, `(<Xn|SP>, (<Wm>|<Xm>)\{, <extend> {<amount>}})`

if option<1> == '0' then UNDEFINED; // sub-word index

**ExtendType** extend_type = DecodeRegExtend(option);

integer shift = if S == '1' then 1 else 0;

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>` When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>` When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>` Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when `<amount>` is omitted, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th><code>&lt;extend&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- `<amount>` Is the index shift amount, optional only when `<extend>` is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th><code>&lt;amount&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

**Shared Decode**

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
  SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(16) data;

if n == 31 then
  CheckSPLignment();
  address = SP[];
else
  address = X[n];
address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>imm9</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<simm>

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>imm9</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<simm>!

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<simm>!

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>imm12</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
32-bit (opc == 11)

LDRSB <Wt>, [<Xn|SP>], #<pimm>]

64-bit (opc == 10)

LDRSB <Xt>, [<Xn|SP>], #<pimm>]

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSB (immediate).

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE; // writeback is suppressed
        when Constraint_UNKNOWN wb unknown = TRUE; // writeback is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLPSL);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN rt unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t];
        Mem[address, 1, AccType_NORMAL] = data;
    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);
    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDRSB (register)**

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

```
0 0 1 1 1 0 0 0 1 x 1  | Rm | option | S | 1 0  | Rn | Rt

size opc
```

### 32-bit with extended register offset (opc == 11 & option != 011)

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]  
```

### 32-bit with shifted register offset (opc == 11 & option == 011)

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  
```

### 64-bit with extended register offset (opc == 10 & option != 011)

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]  
```

### 64-bit with shifted register offset (opc == 10 & option == 011)

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  
```

If option<1> == '0' then UNDEFINED;  // sub-word index

```
ExtendType  extend_type = DecodeRegExtend(option);  
```

**Assembler Symbols**

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>` is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>` is the index extend specifier, encoded in “option”:
  
<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- `<amount>` is the index shift amount, it must be #0, encoded in “S” as 0 if omitted, or as 1 if present.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, 0);
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
  if memop != MemOp_PREFETCH then
    CheckSPAlignment();
    address = SP[];
  else
    address = X[n];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 1, AccType_NORMAL] = data;

  when MemOp_LOAD
    data = Mem[address, 1, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>imm9</td>
<td></td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rt</td>
</tr>
</tbody>
</table>

32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>], #<simm>

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>imm9</td>
<td></td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rt</td>
</tr>
</tbody>
</table>

32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>], #<simm>

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

Unsigned offset

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>imm12</td>
<td></td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rt</td>
</tr>
</tbody>
</table>

LDRSH (immediate)
32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]  

64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]  

boolean wback = FALSE;  
boolean postindex = FALSE;  
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);  

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRSH (immediate).

Assembler Symbols

<Wt>  
Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt>  
Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP>  
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm>  
Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm>  
Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<l> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE;  // writeback is suppressed
    when Constraint_UNKNOWN wb unknown = TRUE;  // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {ConstraintNONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt unknown = FALSE;  // value stored is original value
    when Constraint_UNKNOWN rt unknown = TRUE;  // value stored is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];
if !postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    if rt_unknown then
      data = bits(16) UNKNOWN;
    else
      data = X[t];
    Mem[address, 2, AccType_NORMAL] = data;
  when MemOp_LOAD
    data = Mem[address, 2, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

if wback then
  if wb unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | x  |    | Rm |    | option | S | 1 | 0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**32-bit (opc == 11)**

LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

**64-bit (opc == 10)**

LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

if option<1> == '0' then UNDEFINED; // sub-word index

ExtendType extend_type = DecodeRegExtend(option);

integer shift = if S == '1' then 1 else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
        address = SP[];
    else
        address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset.

### Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| 1 0 1 1 0 0 0 1 0 0 | 0 1 | Rn | Rt |

LDRSW <Xt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

### Pre-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| 1 0 1 1 0 0 0 1 0 0 | 1 1 | Rn | Rt |

LDRSW <Xt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

### Unsigned offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| 1 0 1 1 1 0 0 1 1 0 | imm12 | Rn | Rt |

LDRSW <Xt>, [<Xn|SP>, {, #<pimm>}]

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly LDRSW (immediate).

### Assembler Symbols

- **<Xt>** Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<simm>** Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- **<pimm>** Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
  c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
  assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_WBSUPPRESS wback = FALSE;   // writeback is suppressed
    when Constraint_UNKNOWN wb unknown = TRUE;   // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
endcase;

Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
endif

if !postindex then
  address = address + offset;
endif

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);

if wback then
  if wb_unknown then
    address = bits(64) UNKNOWN;
  elsif postindex then
    address = address + offset;
  endif
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;
endif

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 1 0 0 0 imm19
  opc

LDRSW <Xt>, <label>

integer t = UInt(Rt);
bits(64) offset;
offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(64) address = PC[] + offset;
bits(32) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);
data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | Rm | option | S | 1 | 0 | Rn | | | | | | | | | | | | | | | | | \begin{tabular}{|c|c|}
\hline
size & opc \\
\hline
\end{tabular}\

LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

if option<1> == '0' then UNDEFINED; // sub-word index

ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0;

Assembler Symbols

< Xt > Is the 64-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTW</td>
</tr>
<tr>
<td>001</td>
<td>LSL</td>
</tr>
<tr>
<td>010</td>
<td>SXTW</td>
</tr>
<tr>
<td>011</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(32) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
else
    address = X[n];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSETA and LDSETAL load from memory with acquire semantics.
- LDSETL and LDSETAL store to memory with release semantics.
- LDSET has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSET, STSETL.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

size opc
32-bit LDSET (size == 10 & A == 0 & R == 0)

LDSET <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETA (size == 10 & A == 1 & R == 0)

LDSETA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETAL (size == 10 & A == 1 & R == 1)

LDSETAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSETL (size == 10 & A == 0 & R == 1)

LDSETL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSET (size == 11 & A == 0 & R == 0)

LDSET <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETA (size == 11 & A == 1 & R == 0)

LDSETA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETAL (size == 11 & A == 1 & R == 1)

LDSETAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSETL (size == 11 & A == 0 & R == 1)

LDSETL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSET, STSETL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSETB, STSETLB.

Integer
(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size opc

LDSETAB (A == 1 & R == 0)

LDSETAB <Ws>, <Wt>, [<Xn|SP>]

LDSETALB (A == 1 & R == 1)

LDSETALB <Ws>, <Wt>, [<Xn|SP>]

LDSETB (A == 0 & R == 0)

LDSETB <Ws>, <Wt>, [<Xn|SP>]

LDSETLB (A == 0 & R == 1)

LDSETLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETB, STSETLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
  X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSETH, LDSETHAH, LDSETHALH, LDSETHLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETHAH and LDSETHALH load from memory with acquire semantics.
- LDSETHLH and LDSETHALH store to memory with release semantics.
- LDSETH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSETH, STSETLH.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| size | opc |

LDSETHA (A == 1 && R == 0)

LDSETH <Ws>, <Wt>, [<Xn|SP>]

LDSETHAL (A == 1 && R == 1)

LDSETHAL <Ws>, <Wt>, [<Xn|SP>]

LDSETH (A == 0 && R == 0)

LDSETH <Ws>, <Wt>, [<Xn|SP>]

LDSETHL (A == 0 && R == 1)

LDSETHL <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSETH, STSETLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp.ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL**

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMAXA and LDSMAXAL load from memory with acquire semantics.
- LDSMAXL and LDSMAXAL store to memory with release semantics.
- LDSMAX has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STSMAX, STSMAXL**.

**Integer**  
*Armv8.1*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | Rn |

size opc
32-bit LDSMAX (size == 10 && A == 0 && R == 0)

LDSMAX <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXA (size == 10 && A == 1 && R == 0)

LDSMAXA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXAL (size == 10 && A == 1 && R == 1)

LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMAXL (size == 10 && A == 0 && R == 1)

LDSMAXL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSMAX (size == 11 && A == 0 && R == 0)

LDSMAX <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXA (size == 11 && A == 1 && R == 0)

LDSMAXA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXAL (size == 11 && A == 1 && R == 1)

LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMAXL (size == 11 && A == 0 && R == 1)

LDSMAXL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAX, STSMAXL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMAXB, STSMAXLB.

Integer (Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
</tr>
</tbody>
</table>

size  opc

LDSMAXAB (A == 1 & & R == 0)

LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXALB (A == 1 & & R == 1)

LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXB (A == 0 & & R == 0)

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

LDSMAXLB (A == 0 & & R == 1)

LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' & & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;

boolean tag_checked = n != 31;

Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt>  Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXB_STSMAXLB</td>
<td>A == '0' &amp; &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
  X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMAXH, STSMAXLH.

Integer
(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|--------|
| 0 1 1 1 1 0 0 0 A R 1 | Rs 0 1 0 0 0 0 Rn Rt |

LDSMAXH (A == 1 && R == 0)

LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]

LDSMAXALH (A == 1 && R == 1)

LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]

LDSMAXH (A == 0 && R == 0)

LDSMAXH <Ws>, <Wt>, [<Xn|SP>]

LDSMAXLH (A == 0 && R == 1)

LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMAXH_STSMAXLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDSMINA and LDSMINAL load from memory with acquire semantics.
- LDSMINL and LDSMINAL store to memory with release semantics.
- LDSMIN has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STSMIN, STSMINL**.

**Integer**

(Armv8.1)

31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0

| 1 | x | 1 | 1 | 1 | 0 | 0 | 0 | A | R | 1 | Rs | 0 | 1 | 0 | 1 | 0 | 0 | Rn | Rt |
| size | opc |

LDSMIN, LDSMINA, LDSMINAL, LDSMINL
32-bit LDSMIN (size == 10 && A == 0 && R == 0)
LDSMIN <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINA (size == 10 && A == 1 && R == 0)
LDSMINA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINAL (size == 10 && A == 1 && R == 1)
LDSMINAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDSMINL (size == 10 && A == 0 && R == 1)
LDSMINL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDSMIN (size == 11 && A == 0 && R == 0)
LDSMIN <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINA (size == 11 && A == 1 && R == 0)
LDSMINA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINAL (size == 11 && A == 1 && R == 1)
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDSMINL (size == 11 && A == 0 && R == 1)
LDSMINL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

t = UInt(Rt);
n = UInt(Rn);
s = UInt(Rs);

datasize = 8 << UInt(size);
regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMIN, STSMINL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
  X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMINB, STSMINLB.

**Integer**

(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs  | 0  | 1  | 0  | 1  | 0  | 0  | Rn  | Rt  |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>

**LDSMINB (A == 1 && R == 0)**

LDSMINB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINALB (A == 1 && R == 1)**

LDSMINALB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINB (A == 0 && R == 0)**

LDSMINB <Ws>, <Wt>, [<Xn|SP>]

**LDSMINLB (A == 0 && R == 1)**

LDSMIN LB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMINB, STSMINLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STSMINH, STSMINLH.

**Integer**

(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn | 0  | 1  | 0  | 0  | 0  | A  | R  | 1  | Rs |
| size | opc |

**LDSMINH (A == 1 && R == 0)**

LDSMINH <Ws>, <Wt>, [<Xn|SP>]

**LDSMINAH (A == 1 && R == 1)**

LDSMINAH <Ws>, <Wt>, [<Xn|SP>]

**LDSMINALH (A == 0 && R == 0)**

LDSMINALH <Ws>, <Wt>, [<Xn|SP>]

**LDSMINLH (A == 0 && R == 1)**

LDSMINLH <Ws>, <Wt>, [<Xn|SP>]

```java
if !HaveAtomicExt() then UNDEFINED;

t = UInt(Rt);
n = UInt(Rn);
s = UInt(Rs);

AccType ldacctype = if A == '1' & Rt != '11111' then AccType.ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType.ORDEREDATOMICRW else AccType_ATOMICRW;

boolean tag_checked = n != 31;
```

**Assembler Symbols**

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STSMINH_STSMINLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm9</td>
<td></td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDTR <Wt>, [<Xn|SP>{, #<simm}]

64-bit (size == 11)

LDTR <Xt>, [<Xn|SP>{, #<simm}]

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

integer regsize;
regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, acctype];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>

LDTRB <Wt>, [<Xn|SP>{, #<simm}>]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !((EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, acctype];
X[t] = ZeroExtend(data, 32);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>imm9</td>
<td>1</td>
</tr>
</tbody>
</table>

LDTRH <Wt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;
data = Mem[address, 2, acctype];
X[t] = ZeroExtend(data, 32);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:
  • The instruction is executed at EL1.
  • The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.
Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

32-bit (opc == 11)

LDTRSB \(<Wt>, [<Xn|SP>{, #<simm>}])

64-bit (opc == 10)

LDTRSB \(<Xt>, [<Xn|SP>{, #<simm>}])

\[\text{bits}(64) \text{ offset} = \text{SignExtend}(\text{imm9, 64});\]

**Assembler Symbols**

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xt>\) Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{simm}>\) Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
Shared Decode

```c
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;
if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```c
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[0];
else
  address = X[n];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 1, acctype] = data;
  when MemOp_LOAD
    data = Mem[address, 1, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.


**LDTRSH**

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

---

**32-bit (opc == 11)**

LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]

**64-bit (opc == 10)**

LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = \text{SignExtend}(imm9, 64);

---

**Assembler Symbols**

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '1';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '1';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2)
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);

Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 2, acctype] = data;
  when MemOp_LOAD
    data = Mem[address, 2, acctype];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);
  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.E2H, TGE is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>

LDTRSW \(<Xt>, \ [<Xn|SP>{, \ #<simm>}\}

bits(64) offset = \texttt{SignExtend}(imm9, 64);

Assembler Symbols

\(<Xt>\) Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{imm9}>\) Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = \texttt{UInt}(Rn);
integer t = \texttt{UInt}(Rt);

unpriv_at_el1 = \texttt{PSTATE.EL == EL1} && !\texttt{EL2Enabled()} \&\& \texttt{HaveNVExt()} \&\& \texttt{HCR_EL2.<NV,NV1> == '11'};
unpriv_at_el2 = \texttt{PSTATE.EL == EL2} \&\& \texttt{HaveVirtHostExt()} \&\& \texttt{HCR_EL2.<E2H,TGE> == '11'};

user_access_override = \texttt{HaveUAOExt()} \&\& \texttt{PSTATE.UAO == '1'};
if !\texttt{user_access_override} \&\& \texttt{(unpriv_at_el1 || unpriv_at_el2)} then
  acctype = \texttt{AccType_UNPRIV};
else
  acctype = \texttt{AccType_NORMAL};

boolean tag_checked = n != 31;

Operation

if \texttt{HaveMTEExt()} then
  \texttt{SetTagCheckedInstruction}(tag_checked);

bits(64) address;
bits(32) data;

if n == 31 then
  \texttt{CheckSPAlignment();}
  address = \texttt{SP}[];
else
  address = \texttt{X}[n];

address = address + offset;

data = \texttt{Mem}[address, 4, acctype];
\texttt{X}[t] = \texttt{SignExtend}(data, 64);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL**

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, LDUMAXA and LDUMAXAL load from memory with acquire semantics.
- LDUMAXL and LDUMAXAL store to memory with release semantics.
- LDUMAX has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias **STUMAX, STUMAXL**.

**Integer**

*(Armv8.1)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

*opc*
32-bit LDUMAX (size == 10 & A == 0 & R == 0)
LDUMAX <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMAXA (size == 10 & A == 1 & R == 0)
LDUMAXA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMAXAL (size == 10 & A == 1 & R == 1)
LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMAXL (size == 10 & A == 0 & R == 1)
LDUMAXL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDUMAX (size == 11 & A == 0 & R == 0)
LDUMAX <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMAXA (size == 11 & A == 1 & R == 0)
LDUMAXA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMAXAL (size == 11 & A == 1 & R == 1)
LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMAXL (size == 11 & A == 0 & R == 1)
LDUMAXL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);
integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols
<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAX, STUMAXL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMAXB, STUMAXLB.

**Integer**

(Armv8.1)

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**LDUMAXAB (A == 1 && R == 0)**

LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]

**LDUMAXALB (A == 1 && R == 1)**

LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]

**LDUMAXB (A == 0 && R == 0)**

LDUMAXB <Ws>, <Wt>, [<Xn|SP>]

**LDUMAXLB (A == 0 && R == 1)**

LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXB_STUMAXLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release.
For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 1  | 0  | 0  | 0  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

size opc

LDUMAXH (A == 1 & R == 0)

LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]

LDUMAXALH (A == 1 & R == 1)

LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]

LDUMAXH (A == 0 & R == 0)

LDUMAXH <Ws>, <Wt>, [<Xn|SP>]

LDUMAXLH (A == 0 & R == 1)

LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMAXH_STUMAXLH</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDUMIN, LDUMINA, LDUMINAL, LDUMINL**

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, **LDUMINA** and **LDUMINAL** load from memory with acquire semantics.
- **LDUMINL** and **LDUMINAL** store to memory with release semantics.
- **LDUMIN** has neither acquire nor release semantics.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).
For information about memory accesses see [Load/Store addressing modes](#).

This instruction is used by the alias **STUMIN, STUMINL**.

**Integer**  
*(Armv8.1)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  |  x | 1 |  1 |  1 |  0 |  0 |  0 |  A |  R |  1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
</table>

[Load-Acquire, Store-Release](#): 

[Load/Store addressing modes](#):
Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMIN, STUMINL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

Assembler Symbols

- \(<Ws>\): Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Wt>\): Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- \(<Xs>\): Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xt>\): Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- \(<Xn|SP>\): Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.


**Operation**

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAB and LDUMINALB load from memory with acquire semantics.
- LDUMINLB and LDUMINALB store to memory with release semantics.
- LDUMINB has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMINB, STUMINLB.

Integer (Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | A  | R  | 1  | Rs  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

size opc

**LDUMINAB** (A == 1 && R == 0)

LDUMINAB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINALB** (A == 1 && R == 1)

LDUMINALB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINB** (A == 0 && R == 0)

LDUMINB <Ws>, <Wt>, [<Xn|SP>]

**LDUMINLB** (A == 0 && R == 1)

LDUMINLB <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

_ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

_Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

_Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINB, STUMINLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);
if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

This instruction is used by the alias STUMINH, STUMINLH.

Integer
(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 1 1 0 0 0 A R | 1 | Rs | 0 1 1 1 0 0 | Rn | Rt |

size opc

LDUMINAH (A == 1 && R == 0)

LDUMINAH <Ws>, <Wt>, [<Xn|SP>]

LDUMINALH (A == 1 && R == 1)

LDUMINALH <Ws>, <Wt>, [<Xn|SP>]

LDUMINH (A == 0 && R == 0)

LDUMINH <Ws>, <Wt>, [<Xn|SP>]

LDUMINLH (A == 0 && R == 1)

LDUMINLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUMINH, STUMINLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

value = X[s];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

### 32-bit (size == 10)

LDUR <Wt>, [<Xn|SP>{, #<simm>}]  

### 64-bit (size == 11)

LDUR <Xt>, [<Xn|SP>{, #<simm>}]  

integer scale = UInt(size);  
bits(64) offset = SignExtend(imm9, 64);

#### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xt>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

#### Shared Decode

integer n = UInt(Rn);  
integer t = UInt(Rt);  
integer regsize;

regsize = if size == '11' then 64 else 32;  
integer datasize = 8 << scale;  
boolean tag_checked = n != 31;

#### Operation

if HaveMTEExt() then  
    SetTagCheckedInstruction(tag_checked);

bits(64) address;  
bits(datasize) data;

if n == 31 then  
    CheckSPAlignment();  
    address = SP[];  
else  
    address = X[n];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];  
X[t] = ZeroExtend(data, regsize);
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|--------|--------|----|
|   0  1  1  1  0  0  0  0  1  0   |   imm9 |   0  0 |   Rn |   Rt |
|     size                      |        |      opc
```

LDURB <Wt>, [<Xn|SP>{, #<simm>}

bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```python
integer n = Uint(Rn);
integer t = Uint(Rt);

boolean tag_checked = n != 31;
```

**Operation**

```python
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|
| size                                          | opc           |
```

LDURH <Wt>, [<Xn|SP>{, #<simm>}

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

Operation

```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

32-bit (opc == 11)

LDURSB <Wt>, [<Xn|SP>{, #<simm>}]

64-bit (opc == 10)

LDURSB <Xt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
  // store or zero-extending load
  memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
  regsize = 32;
  signed = FALSE;
else
  // sign-extending load
  memop = MemOp_LOAD;
  regsize = if opc<0> == '1' then 32 else 64;
  signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
  if memop != MemOp_PREFETCH then CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

case memop of
  when MemOp_STORE
    data = X[t];
    Mem[address, 1, AccType_NORMAL] = data;

  when MemOp_LOAD
    data = Mem[address, 1, AccType_NORMAL];
    if signed then
      X[t] = SignExtend(data, regsize);
    else
      X[t] = ZeroExtend(data, regsize);

  when MemOp_PREFETCH
    Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

```
0 1 1 1 0 0 1 0 | imm9 | 0 0 | Rn | Rt
size        opc
```

32-bit (opc == 11)

LDURSH <Wt>, [<Xn|SP>{, #<simm}>]

64-bit (opc == 10)

LDURSH <Xt>, [<Xn|SP>{, #<simm}>]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>  Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```java
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(32) data;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = Mem[address, 4, AccType_NORMAL];
X[t] = SignExtend(data, 64);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit
doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword
aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned
and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being
accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See
Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

32-bit (sz == 0)

LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on
UNPREDICTABLE behaviors, and particularly LDXP.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
data = Mem[address, dbytes, AccType_ATOMIC];
    if BigEndian(AccType_ATOMIC) then
        X[t] = data<datasize-1:elsize>;
        X[t2] = data<elsize-1:0>;
    else
        X[t] = data<elsize-1:0>;
        X[t2] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AArch64.AlignmentFault(AccType_ATOMIC, FALSE, FALSE));
    X[t] = Mem[address, 8, AccType_ATOMIC];
    X[t2] = Mem[address+8, 8, AccType_ATOMIC];

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

32-bit (size == 10)

LDXR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

LDXR <Xt>, [<Xn|SP>{,#0}]

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ATOMIC];
X[t] = ZeroExtend(data, regsize);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 0 0 1 0</td>
</tr>
</tbody>
</table>

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assemble Symbols

<\(W_t\)> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<\(X_n\)|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTEExt() then
SetTagCheckedInstruction(tag_checked);

if n == 31 then
CheckSAPermission();
address = SP[];
else
address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-
extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed
as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and
semaphores. For information about memory accesses see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| 0 1 0 0 1 0 0 0 0 | 1 0 1 0 | 0 (1) (1) (1) (1) | 0 (1) (1) (1) (1) |

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPANalysis();
  address = SP[];
else
  address = X[n];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ATOMIC];
X[t] = ZeroExtend(data, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LSL (register)**

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This is an alias of `LSLV`. This means:

- The encodings in this description are named to match the encodings of `LSLV`.
- The description of `LSLV` gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

```
LSL <Wd>, <Wn>, <Wm>
```

is equivalent to

```
LSLV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

### 64-bit (sf == 1)

```
LSL <Xd>, <Xn>, <Xm>
```

is equivalent to

```
LSLV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

The description of `LSLV` gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 && N == 0 && imms != 011111)

LSL <Wd>, <Wn>, #<shift>

is equivalent to

UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)

and is the preferred disassembly when imms + 1 == immr.

64-bit (sf == 1 && N == 1 && imms != 111111)

LSL <Xd>, <Xn>, #<shift>

is equivalent to

UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)

and is the preferred disassembly when imms + 1 == immr.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <shift> For the 32-bit variant: is the shift amount, in the range 0 to 31.
- For the 64-bit variant: is the shift amount, in the range 0 to 63.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**LSLV**

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias **LSL (register)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

\[
\text{LSLV } <Wd>, <Wn>, <Wm>
\]

**64-bit (sf == 1)**

\[
\text{LSLV } <Xd>, <Xn>, <Xm>
\]

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

**Assembler Symbols**

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of \textit{LSRV}. This means:

- The encodings in this description are named to match the encodings of \textit{LSRV}.
- The description of \textit{LSRV} gives the operational pseudocode for this instruction.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf | 0 0 1 1 0 1 0 1 1 0 | Rm | 0 0 1 0 | 0 1 | Rn | Rd

op2
\end{verbatim}

**32-bit (sf == 0)**

\texttt{LSR <Wd>, <Wn>, <Wm>}

is equivalent to

\texttt{LSRV <Wd>, <Wn>, <Wm>}

and is always the preferred disassembly.

**64-bit (sf == 1)**

\texttt{LSR <Xd>, <Xn>, <Xm>}

is equivalent to

\texttt{LSRV <Xd>, <Xn>, <Xm>}

and is always the preferred disassembly.

**Assembler Symbols**

\begin{itemize}
\item \texttt{<Wd>} Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
\item \texttt{<Wn>} Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
\item \texttt{<Wm>} Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
\item \texttt{<Xd>} Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\item \texttt{<Xn>} Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
\item \texttt{<Xm>} Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.
\end{itemize}

**Operation**

The description of \textit{LSRV} gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>x</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>imms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0 && imms == 011111)

LSR <Wd>, <Wn>, #<shift>

is equivalent to

UBFM <Wd>, <Wn>, #<shift>, #31

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1 && imms == 111111)

LSR <Xd>, <Xn>, #<shift>

is equivalent to

UBFM <Xd>, <Xn>, #<shift>, #63

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <shift> For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field.
  For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias **LSR (register)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (sf == 0)

**LSRV** `<Wd>, <Wn>, <Wm>`

### 64-bit (sf == 1)

**LSRV** `<Xd>, <Xn>, < Xm>`

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer m = UInt(Rm);`
- `integer datasize = if sf == '1' then 64 else 32;`
- `ShiftType shift_type = DecodeShift(op2);`

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL.

<table>
<thead>
<tr>
<th>sf</th>
<th>Rm</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MADD <Wd>, <Wn>, <Wm>, <Wa>

64-bit (sf == 1)

MADD <Xd>, <Xn>, <Xm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa> Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa> Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;

result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d] = result<destsize-1:0>;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode for this instruction.

32-bit (sf == 0)

MNEG \(<Wd>, <Wn>, <Wm>\)

is equivalent to

MSUB \(<Wd>, <Wn>, <Wm>, WZR\)

and is always the preferred disassembly.

64-bit (sf == 1)

MNEG \(<Xd>, <Xn>, < Xm>\)

is equivalent to

MSUB \(<Xd>, <Xn>, < Xm>, XZR\)

and is always the preferred disassembly.

Assembler Symbols

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Wm>\) Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- \(<Xm>\) Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MOV (to/from SP)**

Move between register and stack pointer: \( Rd = Rn \).

This is an alias of **ADD (immediate)**. This means:

- The encodings in this description are named to match the encodings of **ADD (immediate)**.
- The description of **ADD (immediate)** gives the operational pseudocode for this instruction.

| sf | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
| op | S | sh | imm12 |

### 32-bit (sf == 0)

\[
\text{MOV } <\text{Wd}|\text{WSP}>, <\text{Wn}|\text{WSP}>
\]

is equivalent to

\[
\text{ADD } <\text{Wd}|\text{WSP}>, <\text{Wn}|\text{WSP}>, \#0
\]

and is the preferred disassembly when \( (Rd == '11111' || Rn == '11111') \).

### 64-bit (sf == 1)

\[
\text{MOV } <\text{Xd}|\text{SP}>, <\text{Xn}|\text{SP}>
\]

is equivalent to

\[
\text{ADD } <\text{Xd}|\text{SP}>, <\text{Xn}|\text{SP}>, \#0
\]

and is the preferred disassembly when \( (Rd == '11111' || Rn == '11111') \).

**Assembler Symbols**

- `<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of **ADD (immediate)** gives the operational pseudocode for this instruction.
MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This is an alias of MOVN. This means:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 0 0 1 0 1</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
</table>

32-bit (sf == 0 & hw == 0x)

MOV <Wd>, #<imm>

is equivalent to

MOVN <Wd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16).

64-bit (sf == 1)

MOV <Xd>, #<imm>

is equivalent to

MOVN <Xd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<imm> For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff

For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".

<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 & hw == 0x)

MOV <Wd>, #<imm>

is equivalent to

MOVZ <Wd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

64-bit (sf == 1)

MOV <Xd>, #<imm>

is equivalent to

MOVZ <Xd>, #<imm16>, LSL #<shift>

and is the preferred disassembly when ! (IsZero(imm16) && hw != '00').

Assembler Symbols

- **<Wd>**
  - Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

- **<Xd>**
  - Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

- **<imm>**
  - For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw".
    - For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".

- **<shift>**
  - For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
    - For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

32-bit (sf == 0 & N == 0)

MOV <Wd|WSP>, #<imm>

is equivalent to

ORR <Wd|WSP>, WZR, #<imm>

and is the preferred disassembly when \(!\) MoveWidePreferred(sf, N, imms, immr).

64-bit (sf == 1)

MOV <Xd|SP>, #<imm>

is equivalent to

ORR <Xd|SP>, XZR, #<imm>

and is the preferred disassembly when \(!\) MoveWidePreferred(sf, N, imms, immr).

Assembler Symbols

\(<\text{Wd}\text{|WSP}>\quad \text{Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.}\)
\(<\text{Xd}\text{|SP}>\quad \text{Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.}\)
\(<\text{imm}>\quad \text{For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN.}\)
\(<\text{imm}>\quad \text{For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.}\)

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (register)

Move (register) copies the value in a source register to the destination register.

This is an alias of **ORR (shifted register)**. This means:

- The encodings in this description are named to match the encodings of **ORR (shifted register)**.
- The description of **ORR (shifted register)** gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| sf | 0 1 | 0 1 0 0 0 | Rm | 0 0 0 0 0 1 1 1 1 1 | Rd |
| opt | shift | N | imm6 | Rn |

### 32-bit (sf == 0)

MOV `<Wd>`, `<Wm>`

is equivalent to

**ORR `<Wd>`, WZR, `<Wm>`**

and is always the preferred disassembly.

### 64-bit (sf == 1)

MOV `<Xd>`, `<Xm>`

is equivalent to

**ORR `<Xd>`, XZR, `<Xm>`**

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

**Operation**

The description of **ORR (shifted register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MOVK**

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

```
<table>
<thead>
<tr>
<th>sf</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
</table>
```

32-bit (sf == 0 && hw == 0x)

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<imm>** Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
- **<shift>** For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.
  
  For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

**Operation**

```
bits(datasize) result;
result = X[d];
result<pos+15:pos> = imm16;
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 0 0 1 0 0 1 0 1 hw imm16 Rd
```

**Asmblr Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<imm> Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

<shift> For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>64-bit</td>
<td>! (IsZero(imm16) &amp; hw != '00')</td>
</tr>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>32-bit</td>
<td>! (IsZero(imm16) &amp; hw != '00') &amp; ! IsOnes(imm16)</td>
</tr>
</tbody>
</table>

**Operation**

```
bits(datasize) result;
result = Zeros();
result<pos+15:pos> = imm16;
result = NOT(result);
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
**MOVZ**

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias **MOV (wide immediate)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1 0 0 0 0 1 0 1</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0 && hw == 0x)**

MOVZ <Wd>, #<imm>{, LSL #<shift>}

**64-bit (sf == 1)**

MOVZ <Xd>, #<imm>{, LSL #<shift>}

integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;
if sf == '0' & hw<1> == '1' then UNDEFINED;
pos = UInt(hw:’0000’);

**Assembler Symbols**

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<imm>  Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

<shift>  For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16.

For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (wide immediate)</td>
<td>!_IsZero(imm16) &amp; hw != '00'</td>
</tr>
</tbody>
</table>

**Operation**

bits(datasize) result;
result = Zeros();
result<pos+15:pos> = imm16;
X[d] = result;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Move System Register allows the PE to read an AArch64 System register into a general-purpose register.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>o0</td>
<td>op1</td>
<td>CRn</td>
<td>CRm</td>
<td>op2</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

MRS <Xt>, (systemreg|<op0>_<op1>_<Cn>_<Cm>_<op2>)

`AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);`

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".

The System register names are defined in 'AArch64 System Registers' in the System Register XML.

<op0> Is an unsigned immediate, encoded in "o0":

<table>
<thead>
<tr>
<th>o0</th>
<th>&lt;op0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
X[t] = AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
```
MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see *Process state, PSTATE*.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If ARMv8.0-SSBS is implemented, PSTATE.SSBS.
- If ARMv8.1-PAN is implemented, PSTATE.PAN.
- If ARMv8.2-UAO is implemented, PSTATE.UAO.
- If ARMv8.4-DIT is implemented, PSTATE.DIT.
- If ARMv8.5-MemTag is implemented, PSTATE.TCO.
- If FEAT_CSRE is implemented, PSTATE.CSR.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
 1  1  0  1  0  1  0  0  0  0  0  0  0  0  0  0  1  0  0  CRm  0  1  0  0  op1  op2  1  1  1  1  1
```
MSR <pstatefield>, #<imm>
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');
bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
  when '00x'
    min_EL = EL1;
  when '010'
    min_EL = EL1;
  when '011'
    min_EL = EL0;
  when '100'
    min_EL = EL2;
  when '101'
    UNDEFINED;
  when '110'
    min_EL = EL3;
  when '111'
    min_EL = EL1;
    need_secure = TRUE;
if UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && !IsSecure()) then
  UNDEFINED;

PSTATEField field;
case op1:op2 of
  when '000 011'
    if !HaveUAOExt() then
      UNDEFINED;
    field = PSTATEField_UAO;
  when '000 100'
    if !HavePANExt() then
      UNDEFINED;
    field = PSTATEField_PAN;
  when '000 101' field = PSTATEField_SP;
  when '011 010'
    if !HaveDITExt() then
      UNDEFINED;
    field = PSTATEField_DIT;
  when '011 011'
    if CRm == '000x' then
      if !HaveCSRExt() then
        UNDEFINED;
      field = PSTATEField_CSR;
    else
      UNDEFINED;
    when '011 100'
      if !HaveMTEEExt() then
        UNDEFINED;
      field = PSTATEField_TCO;
  when '011 110' field = PSTATEField_DAIFSet;
  when '011 111' field = PSTATEField_DAIFClr;
  when '011 001'
    if !HaveSSBSExt() then
      UNDEFINED;
    field = PSTATEField_SSBS;
  otherwise UNDEFINED;

  // Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
  if ELU.sUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') || SCTLR_EL1.UMA == '0') then
    // AArch64.SystemAccessTrap(EL2, 0x18);
  else
    AArch64.SystemAccessTrap(EL1, 0x18);

Assembler Symbols

<pstatefield> Is a PSTATE field name, encoded in “op1:op2”:

<table>
<thead>
<tr>
<th>op1</th>
<th>op2</th>
<th>&lt;pstatefield&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00x</td>
<td>SEE_PSTATE</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>SEE_PSTATE</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>UAO</td>
<td>FEAT_UAO</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
<td>PAN</td>
<td>FEAT_PAN</td>
</tr>
<tr>
<td>000</td>
<td>101</td>
<td>SPSel</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>11x</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>xxx</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>xxx</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>SSBS</td>
<td>FEAT_SSBS</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>DIT</td>
<td>FEAT_DIT</td>
</tr>
<tr>
<td>011</td>
<td>011</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>100</td>
<td>TCO</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>011</td>
<td>101</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>110</td>
<td>DAIFSet</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>111</td>
<td>DAIFClr</td>
<td></td>
</tr>
<tr>
<td>1xx</td>
<td>xxx</td>
<td>RESERVED</td>
<td></td>
</tr>
</tbody>
</table>

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field.

Operation

case field of

when PSTATEField_SSBS
  PSTATE.SSBS = CRm<0>;
when PSTATEField_SP
  PSTATE.SP = CRm<0>;
when PSTATEField_DAIFSet
  PSTATE.D = PSTATE.D OR CRm<3>;
PSTATE.A = PSTATE.A OR CRm<2>;
PSTATE.I = PSTATE.I OR CRm<1>;
PSTATE.F = PSTATE.F OR CRm<0>;
when PSTATEField_DAIFClr
  PSTATE.D = PSTATE.D AND NOT(CRm<3>);
PSTATE.A = PSTATE.A AND NOT(CRm<2>);
PSTATE.I = PSTATE.I AND NOT(CRm<1>);
PSTATE.F = PSTATE.F AND NOT(CRm<0>);
when PSTATEField_PAN
  PSTATE.PAN = CRm<0>;
when PSTATEField_UAO
  PSTATE.UAO = CRm<0>;
when PSTATEField_DIT
  PSTATE.DIT = CRm<0>;
when PSTATEField_TCO
  PSTATE.TCO = CRm<0>;
when PSTATEField_CSR
  PSTATE.CSR = CRm<0>;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register:

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  10  9  8  7  6  5  4  3  2  1  0
  1  1  0  1  0  1  0  0  0  1  o0  op1  CRn  CRm  op2  Rt  L
```

MSR (<systemreg>|S<op0>_op1_<Cn>_Cm_<op2>), <Xt>

AArch64.CheckSystemAccess ('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);
integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);

Assembler Symbols

- `<systemreg>` is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
- `<op0>` is an unsigned immediate, encoded in "o0":
<table>
<thead>
<tr>
<th>&lt;op0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
- `<op1>` is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- `<Cn>` is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the "CRn" field.
- `<Cm>` is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the "CRm" field.
- `<op2>` is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- `<Xt>` is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG.

| sf | 0 0 0 1 1 0 1 0 0 0 | Rm | 1 | Ra | Rn | Rd |
|----|----------------------|----|----|----|----|
|    | 00                  |

32-bit (sf == 0)

MSUB <Wd>, <Wn>, <Wm>, <Wa>

64-bit (sf == 1)

MSUB <Xd>, <Xn>, < Xm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa> Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNEG</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(destsize) operand1 = X[n];
bits(destsize) operand2 = X[m];
bits(destsize) operand3 = X[a];

integer result;
result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d] = result<destsize-1:0>
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MUL

Multiply: Rd = Rn * Rm.

This is an alias of MADD. This means:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode for this instruction.

32-bit (sf == 0)

MUL <Wd>, <Wn>, <Wm>

is equivalent to

MADD <Wd>, <Wn>, <Wm>, WZR

and is always the preferred disassembly.

64-bit (sf == 1)

MUL <Xd>, <Xn>, <Xm>

is equivalent to

MADD <Xd>, <Xn>, <Xm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MADD gives the operational pseudocode for this instruction.
**MVN**

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>shift</th>
<th>1</th>
<th>Rm</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>imm6</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

MVN <Wd>, <Wm>{, <shift> #<amount>}

is equivalent to

ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}

and is always the preferred disassembly.

**64-bit (sf == 1)**

MVN <Xd>, <Xm>{, <shift> #<amount>}

is equivalent to

ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  - For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of SUB (shifted register). This means:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

$$\text{NEG} <Wd>, <Wm>{, <shift> \#<amount>}$$

is equivalent to

$$\text{SUB} <Wd>, WZR, <Wm>{, <shift> \#<amount>}$$

and is always the preferred disassembly.

### 64-bit (sf == 1)

$$\text{NEG} <Xd>, <Xm>{, <shift> \#<amount>}$$

is equivalent to

$$\text{SUB} <Xd>, XZR, <Xm>{, <shift> \#<amount>}$$

and is always the preferred disassembly.

### Assembler Symbols

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>`: Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>`: Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<shift>`: Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>`: For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

### Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**NEGS**

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of **SUBS (shifted register)**. This means:

- The encodings in this description are named to match the encodings of **SUBS (shifted register)**.
- The description of **SUBS (shifted register)** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</th>
<th>!= 11111</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

NEGS <Wd>, <Wm>{, <shift> #<amount>}

is equivalent to

SUBS <Wd>, WZR, <Wm> {, <shift> #<amount>}

and is always the preferred disassembly.

**64-bit (sf == 1)**

NEGS <Xd>, <Xm>{, <shift> #<amount>}

is equivalent to

SUBS <Xd>, XZR, <Xm> {, <shift> #<amount>}

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wm>` Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xm>` Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.
- `<shift>` Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<amount>` For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Operation**

The description of **SUBS (shifted register)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.

32-bit (sf == 0)

NGC <Wd>, <Wm>

is equivalent to

SBC <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

NGC <Xd>, <Xm>

is equivalent to

SBC <Xd>, XZR, <Xm>

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SBCS. This means:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode for this instruction.

| sf | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Rd |
|----|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|
| op | S |

32-bit (sf == 0)

NGCS <Wd>, <Wm>

is equivalent to

SBCS <Wd>, WZR, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

NGCS <Xd>, <Xm>

is equivalent to

SBCS <Xd>, XZR, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

NOP

// Empty.

// do nothing

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN.

32-bit (sf == 0)

ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

ORN <Xd>, <Xn>, < Xm>{, <shift> #<amount>}

Assembler Symbols

| <Wd>   | Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Wn>   | Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| <Wm>   | Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <Xd>   | Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field. |
| <Xn>   | Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field. |
| < Xm>  | Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift": |

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

| <amount>  | For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field. |
|           | For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field, |

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) operand1 = $X[n]$;
bits(datasize) operand2 = $\text{ShiftReg}(m, \text{shift\_type}, \text{shift\_amount})$;
operand2 = NOT(operand2);
result = operand1 OR operand2;
$X[d] = \text{result}$;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 Xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ORR (immediate)**

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias **MOV (bitmask immediate)**.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| sf  | 0   | 1   | 0   | 0   | 0   | 0   | N   | immr| immr| imms| Rn  | Rd  |

**32-bit (sf == 0 & N == 0)**

ORR <Wd|WSP>, <Wn>, #<imm>

**64-bit (sf == 1)**

ORR <Xd|SP>, <Xn>, #<imm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' & N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE);

**Assembler Symbols**

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".

For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOV (bitmask immediate)</strong></td>
<td>Rn == '1111' &amp; MoveWidePreferred(sf, N, imms, immr)</td>
</tr>
</tbody>
</table>

**Operation**

bits(datasize) result;
bits(datasize) operand1 = X[n];

result = operand1 OR imm;
if d == 31 then
  SP[] = result;
else
  X[d] = result;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias **MOV (register)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

**ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

64-bit (sf == 1)

**ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

**ShiftType** shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);

Assembler Symbols

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<shift>** Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

**Alias** | Is preferred when
---|---
**MOV (register)** | shift == "00" && imm6 == "000000" && Rn == "11111"
Operation

\[
\begin{align*}
\text{bits(datasize)} \text{ operand1} &= X[n]; \\
\text{bits(datasize)} \text{ operand2} &= \text{ShiftReg}(m, \text{shift\_type}, \text{shift\_amount}); \\
\text{result} &= \text{operand1 OR operand2}; \\
X[d] &= \text{result};
\end{align*}
\]

Operational information

If \text{PSTATE.DIT} is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the \text{NZCV} flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the \text{NZCV} flags.
PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.
The address is in the general-purpose register that is specified by <Xd>.
The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

Integer
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | Z  | 0  | 1  | 0  | Rn |   |   |   |   |   |

PACDA (Z == 0)

PACDA <Xd>, <Xn|SP>

PACDZA (Z == 1 & Rn == 1111)

PACDZA <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == ‘0’ then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UNDEFINED;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the “Rn” field.

Operation

if source_is_sp then
    \text{X}[d] = \text{AddPACDA}(X[d], \text{SP}[\]);
else
    \text{X}[d] = \text{AddPACDA}(X[d], X[n]);
PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.
The address is in the general-purpose register that is specified by <Xd>.
The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

Integer (Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z | 0 | 1 | 1 | Rn | Rd |

PACDB (Z == 0)

PACDB <Xd>, <Xn|SP>

PACDZB (Z == 1 && Rn == 1111)

PACDZB <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UNDEFINED;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if source_is_sp then
    X[d] = AddPACDB(X[d], SP[]);
else
    X[d] = AddPACDB(X[d], X[n]);
PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

Integer

( Armv8.3 )

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  |

PACGA <Xd>, <Xn>, <Xm|SP>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
    UNDEFINED;
if m == 31 then source_is_sp = TRUE;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

if source_is_sp then
    X[d] = AddPACGA(X[n], SP[{}]);
else
    X[d] = AddPACGA(X[n], X[m]);
PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:
- In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

It has encodings from 2 classes: Integer and System

### Integer
(Armv8.3)

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**PACIA (Z == 0)**

PACIA <Xd>, <Xn|SP>

**PACIZA (Z == 1 & Rn == 1111)**

PACIZA <Xd>

```java
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then UNDEFINED;
```

### System
(Armv8.3)

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

CRm op2
PACIA1716 (CRm == 0001 && op2 == 000)
PACIA1716

PACIASP (CRm == 0011 && op2 == 001)
PACIASP

PACIAZ (CRm == 0011 && op2 == 000)
PACIAZ

integer d;
integer n;
boolean source_is_sp = FALSE;
case CRm:op2 of
  when '0011 000'    // PACIAZ
    d = 30;
    n = 31;
  when '0011 001'    // PACIASP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt() then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIASP instruction.
      SetBTypeCompatible(BTypeCompatible_PACIXSP());
  for when '0001 000'    // PACIA1716
    d = 17;
    n = 16;
  when '0001 010' SEE "PACIB";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 01x' SEE "PACIB";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";
otherwise SEE "HINT";

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt() then
  if source_is_sp then
    X[d] = AddPACIA(X[d], SP());
  else
    X[d] = AddPACIA(X[d], X[n]);
PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:
- In the general-purpose register that is specified by <Xd> for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

The modifier is:
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

It has encodings from 2 classes: Integer and System

Integer
(Armv8.3)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PACIB (Z == 0)

PACIB <Xd>, <Xn|SP>

PACIZB (Z == 1 && Rn == 1111)

PACIZB <Xd>

boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;
if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UNDEFINED;

System
(Armv8.3)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

CRm | op2
PACIB1716 (CRm == 0001 && op2 == 010)

PACIB1716

PACIBSP (CRm == 0011 && op2 == 011)

PACIBSP

PACIBZ (CRm == 0011 && op2 == 010)

PACIBZ

integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
  when '0011 010'    // PACIBZ
    d = 30;
    n = 31;
  when '0011 011'    // PACIBSP
    d = 30;
    source_is_sp = TRUE;
    if HaveBTIExt() then
      // Check for branch target compatibility between PSTATE.BTYPE
      // and implicit branch target of PACIBSP instruction.
      SetBTypeCompatible(BTypeCompatible_PACIXSP());
  when '0001 010'    // PACIB1716
    d = 17;
    n = 16;
  when '0001 000' SEE "PACIA";
  when '0001 100' SEE "AUTIA";
  when '0001 110' SEE "AUTIB";
  when '0011 00x' SEE "PACIA";
  when '0011 10x' SEE "AUTIA";
  when '0011 11x' SEE "AUTIB";
  when '0000 111' SEE "XPACLRI";
  otherwise SEE "HINT";

Assembler Symbols

<Xd>    Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

if HavePACExt() then
  if source_is_sp then
    X[d] = AddPACIB(X[d], SP[]);
  else
    X[d] = AddPACIB(X[d], X[n]);
PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory. For information about memory accesses, see Load/Store addressing modes.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
|   |   | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | imm12 |     | Rn  |     | Rt  |
```

PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm}]}

```
bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

- **PLD**
  - Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

- **PLI**
  - Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

- **PST**
  - Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

- **L1**
  - Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

- **L2**
  - Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

- **L3**
  - Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

- **KEEP**
  - Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

- **STRM**
  - Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

bits(64) address;
if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

integer t = Uint(Rt);
bias(64) offset;
offset = SignExtend(imm19:'00', 64);

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.
<type> is one of:
PLD Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
PLI Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
PST Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.
<target> is one of:
L1 Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
L2 Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
L3 Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.
<policy> is one of:
KEEP Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
STRM Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.
Operation

```c
bits(64) address = PC[] + offset;
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);
Prefetch(address, t<4:0>);
```

PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory.

For information about memory accesses, see Load/Store addressing modes.

```
|   1   |
|---|---|---|---|---|---|---|---|---|---|
|   1   |
|---|---|---|---|---|---|---|---|---|---|
|   1   |
|---|---|---|---|---|---|---|---|---|---|
|   1   |
|---|---|---|---|---|---|---|---|---|---|
|   1   |
|---|---|---|---|---|---|---|---|---|---|
|   1   |
|---|---|---|---|---|---|---|---|---|---|
|   1   |
|---|---|---|---|---|---|---|---|---|---|

size opc

PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend} {<amount>}}]
```

if option<1> == '0' then UNDEFINED; // sub-word index

```c
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0;
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

- <type> is one of:
  - PLD
    - Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
  - PLI
    - Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
  - PST
    - Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

- <target> is one of:
  - L1
    - Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
  - L2
    - Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
  - L3
    - Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

- <policy> is one of:
  - KEEP
    - Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
  - STRM
    - Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

For other encodings of the "Rm" field, use <imm5>.
<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>

**Shared Decode**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

**Operation**

```plaintext
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

bits(64) address;
if n == 31 then
    address = SP[{}];
else
    address = X[n];

address = address + offset;
Prefetch(address, t<4:0>);
```
Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see Prefetch memory. For information about memory accesses, see Load/Store addressing modes.

For information about memory accesses, see Load/Store addressing modes.

### Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. 

- **PLD**: Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.
- **PLI**: Preload instructions, encoded in the "Rt<4:3>" field as 0b01.
- **PST**: Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

- **L1**: Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.
- **L2**: Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.
- **L3**: Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

- **KEEP**: Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.
- **STRM**: Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see Prefetch memory. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

### Shared Decode

```c
integer n = UInt(Rn);
integer t = UInt(Rt);
```

```c
size opc
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>})

bits(64) offset = SignExtend(imm9, 64);
```
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(FALSE);

bits(64) address;

if n == 31 then
    address = SP[];
else
    address = X[n];

address = address + offset;

Prefetch(address, t<4:0>);
PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

System
(Armv8.2)

if !HaveStatisticalProfiling() then EndOfInstruction();

Operation

ProfilingSynchronizationBarrier();

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.

- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

\[
\begin{array}{cccccccccccccccccccc}
\hline
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

CRm opc

// No additional decoding required

Operation

SpeculativeStoreBypassBarrierToPA();

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
RBIT

Reverse Bits reverses the bit order in a register.

32-bit (sf == 0)

RBIT <Wd>, <Wn>

64-bit (sf == 1)

RBIT <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(datasize) operand = X[n];
bits(datasize) result;
for i = 0 to datasize-1
    result<datasize-1-i> = operand<i>;
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**RET**

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

\[
\begin{array}{cccccccccccccccccccc}
\hline
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Rn & 0 & 0 & 0 & 0 & 0 & Rm
\end{array}
\]

\[\text{RET } \{<Xn>\}\]

integer \(n = \text{UInt}(\text{Rn});\)

**Assembler Symbols**

\(<Xn>\) Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the “Rn” field. Defaults to X30 if absent.

**Operation**

bits(64) \(\text{target} = X[n];\)

if \(\text{HaveCSRExt}() \&\& \text{CallStackRecordingActive}()\) then
  \(\text{RemoveCallStackRecord}();\)

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

\(\text{BranchTo(target, BranchType\_RET);}\)

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**RETA A, RETAB**

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

Key A is used for RETAA, and key B is used for RETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to LR.

---

**Integer**

*(Armv8.3)*

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | M  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
```

**RETA A (M == 0)**

RETA A

**RETA B (M == 1)**

RETA B

```java
boolean use_key_a = (M == '0');
if !HavePACExt() then
    UNDEFINED;
```

**Operation**

```
bits(64) target = X[30];
bits(64) modifier = SP[];
if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);
if HaveCSRExt() && CallStackRecordingActive() then
    RemoveCallStackRecord();
// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';
BranchTo(target, BranchType_RET);
```
REV

Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | x  | Rn | Rd |

**32-bit (sf == 0 & opc == 10)**

REV <Wd>, <Wn>

**64-bit (sf == 1 & opc == 11)**

REV <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;
X[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**REV16**

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>Rd</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

REV16 <Wd>, <Wn>

**64-bit (sf == 1)**

REV16 <Xd>, <Xn>

```plaintext
text
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
  when '00' Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

```plaintext
bits(datasize) operand = X[n];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
    rev_index = rev_index - 8;
X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

REV32 <Xd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
integer container_size;
case opc of
  when '00'
    Unreachable();
  when '01'
    container_size = 16;
  when '10'
    container_size = 32;
  when '11'
    if sf == '0' then UNDEFINED;
    container_size = 64;

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

bits(datasize) operand = X[n];
bits(datasize) result;
integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
icontainer rev_index;
for c = 0 to containers-1
  rev_index = index + ((elements_per_container - 1) * 8);
  for e = 0 to elements_per_container-1
    result<rev_index+7:rev_index> = operand<index+7:index>;
    index = index + 8;
rev_index = rev_index - 8;
X[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register. When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is optional whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This is a pseudo-instruction of REV. This means:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Rn   | Rd
sf opc

64-bit

REV64 <Xd>, <Xn>

is equivalent to

REV <Xd>, <Xn>

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer (Armv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | imm6 | 0  | 0  | 0  | 0  | 1  | Rn | 0  | mask |

sf

RMIF <Xn>, #<shift>, #<mask>

if !HaveFlagManipulateExt() then UNDEFINED;
integer lsb = UInt(imm6);
integer n = UInt(Rn);

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift> Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
<mask> Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

bits(4) tmp;
bits(64) tmpreg = X[n];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode for this instruction.

| sf | 0 0 1 1 1 | N 0 | Rm | imms | Rn | Rd |

32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

ROR <Wd>, <Ws>, #<shift>

is equivalent to

EXTR <Wd>, <Ws>, <Ws>, #<shift>

and is the preferred disassembly when Rn == Rm.

64-bit (sf == 1 && N == 1)

ROR <Xd>, <Xs>, #<shift>

is equivalent to

EXTR <Xd>, <Xs>, <Xs>, #<shift>

and is the preferred disassembly when Rn == Rm.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Ws> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xs> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
- <shift> For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field.
  For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.

32-bit (sf == 0)

ROR <Wd>, <Wn>, <Wm>

is equivalent to

RORV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

ROR <Xd>, <Xn>, <Xm>

is equivalent to

RORV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register).

32-bit (sf == 0)

RORV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

RORV <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

bits(datasize) result;
bits(datasize) operand2 = X[m];
result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize);
X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SB

Speculation Barrier is a barrier that controls speculation. The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

```
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 1 |(0) (0) (0) |1 1 1 1 1 1 1
```

```
SB
if !HaveSBExt() then UNDEFINED;
```

**Operation**

```
SpeculationBarrier();
```
SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC.

<table>
<thead>
<tr>
<th>sf</th>
<th>1 0 1 0 1 0 0 0 0</th>
<th>Rm</th>
<th>0 0 0 0 0 0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

32-bit (sf == 0)

SBC <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SBC <Xd>, <Xn>, <Xm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGC</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

```plaintext
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS.

<table>
<thead>
<tr>
<th>sf</th>
<th>Rm</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

SBCS <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SBCS <Xd>, <Xn>, < Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

< Wd >  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
< Wn >  Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
< Wm >  Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
< Xd >  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
< Xn >  Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
< Xm >  Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NGCS</td>
<td>Rn == '1111'</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
bits(4) nzcv;
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 && N == 0)

SBFIZ <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 && N == 1)

SBFIZ <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)

and is the preferred disassembly when UInt(imms) < UInt(immr).

Assembler Symbols

<Wr>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wr>  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wr>  Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wr>  Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb> For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31.
<lsb> For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
<width> For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly. If \(<\text{imms}>\) is greater than or equal to \(<\text{immr}>\), this copies a bitfield of \((<\text{imms}>-<\text{immr}>+1)\) bits starting from bit position \(<\text{immr}>\) in the source register to the least significant bits of the destination register. If \(<\text{imms}>\) is less than \(<\text{immr}>\), this copies a bitfield of \((<\text{imms}>+1)\) bits from the least significant bits of the source register to bit position \((\text{regsize}-<\text{immr}>)\) of the destination register, where \text{regsize} is the destination register size of 32 or 64 bits. In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0 && N == 0)

SBFM \(<\text{Wd}>, <\text{Wn}>, #<\text{immr}>, #<\text{imms}>\)

64-bit (sf == 1 && N == 1)

SBFM \(<\text{Xd}>, <\text{Xn}>, #<\text{immr}>, #<\text{imms}>\)

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer R;
integer S;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

R = UInt(immr);
S = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

<immr> For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

<imms> For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR (immediate)</td>
<td>32-bit</td>
<td>imms == '011111'</td>
</tr>
</tbody>
</table>

SBFM
Alias | Of variant | Is preferred when
--- | --- | ---
**ASR (immediate)** | 64-bit | `imms == '111111'`
**SBFIZ** | `UInt(imms) < UInt(immr)` |  
**SBFX** | `BFXPreferred(sf, opc<1>, imms, immr)` |  
**SXTB** | `immr == '000000' && imms == '000111'` |  
**SXTH** | `immr == '000000' && imms == '001111'` |  
**SXTW** | `immr == '000000' && imms == '011111'` |  

### Operation

```plaintext
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// determine extension bits (sign, zero or dest register)
bits(datasize) top = Replicate(src<S>);

// combine extension bits and result bits
X[d] = (top AND NOT(tmask)) OR (bot AND tmask);
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SBFX

Signed Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 & N == 0)

SBFX <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

SBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

64-bit (sf == 1 & N == 1)

SBFX <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

SBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb> For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.

32-bit (sf == 0)

SDIV <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SDIV <Xd>, <Xn>, < Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;
if IsZero(operand2) then
  result = 0;
else
  result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));
X[d] = result<datasize-1:0>;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SETF8, SETF16**

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value.

The PSTATE.C flag is not affected by these instructions.

**Integer**

(Armv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |

**sf**

**SETF8** (*sz == 0*)

**SETF8** `<Wn>`

**SETF16** (*sz == 1*)

**SETF16** `<Wn>`

**Assembler Symbols**

 `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

```plaintext
bits(32) tmpreg = X[n];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb + 1)) then '1' else '0';
PSTATE.V = tmpreg<msb+1> EOR tmpreg<msb>;
//PSTATE.C unchanged;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event*.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 1 1 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 1 1 1 1

SEV

// Empty.

Operation

SendEvent();
```
Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a WFE instruction.

SEVL

CRm  op2

Operation

SendEventLocal();
SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  |

SMADDL <Xd>, <Wn>, <Wm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- **<Xa>** Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMULL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;
result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));

X[d] = result<63:0>;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMC

Secure Monitor Call causes an exception to EL3.
SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.
If the values of HCR_EL2.TSC and SCR_EL3.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in ESR_ELx, using the EC value 0x17, that is taken to EL3.
If the value of HCR_EL2.TSC is 1 and EL2 is enabled in the current Security state, execution of an SMC instruction at EL1 generates an exception that is taken to EL2, regardless of the value of SCR_EL3.SMD. For more information, see Traps to EL2 of Non-secure EL1 execution of SMC instructions.
If the value of HCR_EL2.TSC is 0 and the value of SCR_EL3.SMD is 1, the SMC instruction is UNDEFINED.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  |  | imm16 |  | 0  | 0  | 0  | 0  | 1  | 1  |
```

SMC #<imm>

// Empty.

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the “imm16” field.

Operation

```
AArch64.CheckForSMCUndefOrTrap(imm16);
AArch64.CallSecureMonitor(imm16);
```
SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of SMSUBL. This means:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode for this instruction.

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
   | 0 0 1 1 0 1 1 0 0 1 | Rm | 1 1 1 1 1 1 | Rn  | Rd |
   | U 0 0 | Ra |

SMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

SMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

SMSUBL <Xd>, <Wn>, <Wm>, <Xa>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa> Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMNEGL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];
integer result;
result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d] = result<63:0>;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 0  | 0  | 1  | 1  | 0  | 1  | 0  | Rm | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | Ra |

SMULH <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, FALSE) * Int(operand2, FALSE);

X[d] = result<127:64>;

Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

SMULL <Xd>, <Wn>, <Wm>

is equivalent to

SMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 1 1 1 1 1
```

SSBB

// No additional decoding required

**Operation**

`SpeculativeStoreBypassBarrierToVA();`

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST2G

Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

**Post-index**
**(Armv8.5)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | imm9 | 0  | 1  | Xn | Xt |

ST2G \(<Xt|SP>, [<Xn|SP>], \#, <simm>\)

if !HaveMTEEext() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

**Pre-index**
**(Armv8.5)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | imm9 | 1  | 1  | Xn | Xt |

ST2G \(<Xt|SP>, [<Xn|SP>], \#, <simm>\)

if !HaveMTEEext() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

**Signed offset**
**(Armv8.5)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | imm9 | 1  | 0  | Xn | Xt |

ST2G \(<Xt|SP>, [<Xn|SP>{, \#, <simm>}\)}

if !HaveMTEEext() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

**Assembler Symbols**

\(<Xt|SP>\) Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

\(<\text{simm}>\) Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
ST64B

Single-copy Atomic 64-byte Store without Return stores eight 64-bit doublewords from consecutive registers, \(X_t\) to \(X(t+7)\), to a memory location. The data that is stored is atomic and is required to be 64-byte-aligned.

**Integer**

(Armv8.7)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  |

**Assembler Symbols**

\(<X_t>\) Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<X_n|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

\(\text{CheckLDST64BEnabled}()\);

bits(512) data;
bits(64) address;
bits(64) value;
acctype = \(\text{AccType\_ATOMICLS64}\);

if \(\text{HaveMTEExt}()\) then
    \(\text{SetTagCheckedInstruction}(\text{tag\_checked})\);

for \(i = 0\) to \(7\)
    \(\text{value} = X[t+i]\);
    if \(\text{BigEndian}(\text{acctype})\) then \(\text{value} = \text{BigEndianReverse}(\text{value})\);
    data\:<63+64*i:64*i> = value;

if \(n == 31\) then
    \(\text{CheckSPAlignment}()\);
    address = \(SP[\]\);
else
    address = \(X[n]\);

\(\text{MemStore64B}(\text{address}, \text{data}, \text{acctype})\);
**ST64BV**

Single-copy Atomic 64-byte Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

### Integer

*Armv8.7*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rs | 1  | 0  | 1  | 1  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |

\[ \text{ST64BV} \ <Xs>, \ <Xt>, \ [<Xn|SP>] \]

if !\text{HaveFeatLS64}() \ then \ UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' \ then \ UNDEFINED;

integer n = \text{UInt}(Rn);
integer t = \text{UInt}(Rt);
integer s = \text{UInt}(Rs);
boolean tag_checked = n != 31;

### Assembler Symbols

\(<Xs>\)  
Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.
The value returned is:
\[ 0 \]
  If the operation updates memory.
\[ 1 \]
  If the operation fails to update memory.

\textbf{0xFFFFFFFF_FFFFFFFF}
If the memory location accessed does not support this instruction.
If XZR is used, then the return value is ignored.

\(<Xt>\)  
Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\)  
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

```plaintext
CheckST64BVEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
    value = X[t+i];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
else
    address = X[n];

status = MemStore64BWithRet(address, data, acctype);

if s != 31 then X[s] = status;
```

ST64BV0

Single-copy Atomic 64-byte EL0 Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, with the bottom 32 bits taken from ACCDATA_EL1, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer
(Armv8.7)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | Rs |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Rn |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

ST64BV0 <Xs>, <Xt>, [Xn|SP>]

if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;

Assembler Symbols

<Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.
The value returned is:
0 If the operation updates memory.
1 If the operation fails to update memory.
0xFFFFFFFF_FFFFFFFF If the memory location accessed does not support this instruction.

If XZR is used, then the return value is ignored.

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

```
CheckST64BV0Enabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
    value = X[t+i];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

data<31:0> = ACCDATA_EL1<31:0>;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

status = MemStore64BWithRet(address, data, acctype);

if s != 31 then X[s] = status;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADD does not have release semantics.
- STADDL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

**Integer**

(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| size | A | opc | Rn | Rs | R |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit LDADD alias (size == 10 && R == 0)

STADD <Ws>, [<Xn|SP>]

is equivalent to

LDADD <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

32-bit LDADDL alias (size == 10 && R == 1)

STADDL <Ws>, [<Xn|SP>]

is equivalent to

LDADDL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDADD alias (size == 11 && R == 0)

STADD <Xs>, [<Xn|SP>]

is equivalent to

LDADD <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDADDL alias (size == 11 && R == 1)

STADDL <Xs>, [<Xn|SP>]

is equivalent to

LDADDL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDI, gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADDB, STADDBL

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDB does not have release semantics.
- STADDBL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| size | A  | opc | Rn | 1  | 1  | 1  | 1  | 1  | 1  |

No memory ordering (R == 0)

STADDB <Ws>, [<Xn|SP>]

is equivalent to

LDADDB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STADDBL <Ws>, [<Xn|SP>]

is equivalent to

LDADDLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH does not have release semantics.
- STADDLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |

No memory ordering (R == 0)

STADDH <Ws>, [<Xn|SP>]

is equivalent to

LDADDH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STADDLH <Ws>, [<Xn|SP>]

is equivalent to

LDADDLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws>      Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>   Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit
doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and
stores the result back to memory.

- STCLR does not have release semantics.
- STCLRL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this
  instruction.

**Integer**

(ARMv8.1)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | x | 1 | 1 | 1 | 0 | 0 | 0 | 0 | R | 1 | Rs | 0 | 0 | 0 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | 1 |
| size | A | opc | Rt |

**32-bit LDCLR alias (size == 10 && R == 0)**

STCLR <Ws>, [<Xn|SP>]

is equivalent to

LDCLR <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**32-bit LDCLRL alias (size == 10 && R == 1)**

STCLRL <Ws>, [<Xn|SP>]

is equivalent to

LDCLRL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDCLR alias (size == 11 && R == 0)**

STCLR <Xs>, [<Xn|SP>]

is equivalent to

LDCLR <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDCLRL alias (size == 11 && R == 1)**

STCLRL <Xs>, [<Xn|SP>]

is equivalent to

LDCLRL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STCLRB, STCLRLB**

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise
AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRB does not have release semantics.
- STCLRLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB**. This means:

- The encodings in this description are named to match the encodings of **LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB**.
- The description of **LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB** gives the operational pseudocode for this instruction.

### Integer

(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**No memory ordering (R == 0)**

**STCLRB** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDCLRB** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

**Release (R == 1)**

**STCLRLB** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDCLRLB** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH does not have release semantics.
- STCLRLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH. This means:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

No memory ordering (R == 0)

STCLRH <Ws>, [<Xn|SP>]

is equivalent to

LDCLRH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STCLRLH <Ws>, [<Xn|SP>]

is equivalent to

LDCLRLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEOR does not have release semantics.
- STEORL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDEOR, LDEORA, LDEORAL, LDEORL. This means:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

<table>
<thead>
<tr>
<th>32-bit LDEOR alias (size == 10 &amp; R == 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEOR &lt;Ws&gt;, [&lt;Xn</td>
</tr>
<tr>
<td>is equivalent to</td>
</tr>
<tr>
<td>LDEOR &lt;Ws&gt;, WZR, [&lt;Xn</td>
</tr>
<tr>
<td>and is always the preferred disassembly.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32-bit LDEORL alias (size == 10 &amp; R == 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORL &lt;Ws&gt;, [&lt;Xn</td>
</tr>
<tr>
<td>is equivalent to</td>
</tr>
<tr>
<td>LDEORL &lt;Ws&gt;, WZR, [&lt;Xn</td>
</tr>
<tr>
<td>and is always the preferred disassembly.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit LDEOR alias (size == 11 &amp; R == 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEOR &lt;Xs&gt;, [&lt;Xn</td>
</tr>
<tr>
<td>is equivalent to</td>
</tr>
<tr>
<td>LDEOR &lt;Xs&gt;, XZR, [&lt;Xn</td>
</tr>
<tr>
<td>and is always the preferred disassembly.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit LDEORL alias (size == 11 &amp; R == 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEORL &lt;Xs&gt;, [&lt;Xn</td>
</tr>
<tr>
<td>is equivalent to</td>
</tr>
<tr>
<td>LDEORL &lt;Xs&gt;, XZR, [&lt;Xn</td>
</tr>
<tr>
<td>and is always the preferred disassembly.</td>
</tr>
</tbody>
</table>
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORI, gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORB does not have release semantics.
- STEORLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No memory ordering (R == 0)

STEORB <Ws>, [<Xn|SP>]

is equivalent to

LDEORB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STEORLB <Ws>, [<Xn|SP>]

is equivalent to

LDEORLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

Ws Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- STEORH does not have release semantics.
- STEORLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH. This means:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

**Integer**
(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size A opc Rn Rt

No memory ordering (R == 0)

STEORH <Ws>, [<Xn|SP>]

is equivalent to

LDEORH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STEORLH <Ws>, [<Xn|SP>]

is equivalent to

LDEORLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: **Post-index**, **Pre-index** and **Signed offset**

**Post-index** *(Armv8.5)*

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 1 0 0 1 0 0 1 imm9 0 1 Xn Xt
```

```
STG <Xt|SP>, [<Xn|SP>], #<simm>

integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

**Pre-index** *(Armv8.5)*

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 1 0 0 1 0 0 1 imm9 1 1 Xn Xt
```

```
STG <Xt|SP>, [<Xn|SP>], #<simm>!

integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

**Signed offset** *(Armv8.5)*

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 1 0 0 1 0 0 1 imm9 1 0 Xn Xt
```

```
STG <Xt|SP>, [<Xn|SP>{, #<simm>}]

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

**Assembler Symbols**

- `<Xt|SP>` Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
- `<simm>` Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
STGM

Store Tag Multiple writes a naturally aligned block of N Allocation Tags, where the size of N is identified in
GMID_EL1.BS, and the Allocation Tag written to address A is taken from the source register at
This instruction is UNDEFINED at EL0.
This instruction generates an Unchecked access.
If ID_AA64FPFR1_EL1.MTE != 0b0010, this instruction is UNDEFINED.

Integer
(Armv8.5)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STGM <Xt>, [<Xn|SP>]

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

if PSTATE.EL == EL0 then
  UNDEFINED;
bits(64) data = X[t];
bits(64) address;
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
integer size = 4 * (UInt(GMID_EL1.BS));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);
for i = 0 to count-1
  bits(4) tag = data<(index*4)+3:index*4>;
  AArch64_MemTag[address, AccType_NORMAL] = tag;
  address = address + TAG_GRANULE;
  index = index + 1;
Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

### Post-index

(Armv8.5)

|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |   |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|   |
| 31    | 30    | 29    | 28    | 27    | 26    | 25    | 24    | 23    | 22    | 21    | 20    | 19    | 18    | 17    | 16    | 15    | 14    | 13    | 12    | 11    | 10    |   |
| 1      | 0      | 1      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      |   |
| simm7  | Xt2    | Xn     | Xt     |

STGP `<Xt1>, <Xt2>, [<Xn|SP>], #<imm>`

integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

### Pre-index

(Armv8.5)

|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |   |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|   |
| 31    | 30    | 29    | 28    | 27    | 26    | 25    | 24    | 23    | 22    | 21    | 20    | 19    | 18    | 17    | 16    | 15    | 14    | 13    | 12    | 11    | 10    |   |
| 1      | 0      | 1      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      |   |
| simm7  | Xt2    | Xn     | Xt     |

STGP `<Xt1>, <Xt2>, [<Xn|SP>, #<imm>]`

integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

### Signed offset

(Armv8.5)

|       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |   |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|   |
| 31    | 30    | 29    | 28    | 27    | 26    | 25    | 24    | 23    | 22    | 21    | 20    | 19    | 18    | 17    | 16    | 15    | 14    | 13    | 12    | 11    | 10    |   |
| 1      | 0      | 1      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      | 0      | 1      | 0      |   |
| simm7  | Xt2    | Xn     | Xt     |

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

### Assembler Symbols

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.
<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
For the post-index and pre-index variant: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field.

For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

Operation

```plaintext
bits(64) address;
b bits(64) data1;
b bits(64) data2;
SetTagCheckedInstruction(FALSE);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data1 = X[t];
data2 = X[t2];
if !postindex then
    address = address + offset;
Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;
AArch64.MemTag[address, AccType_NORMAL] = AArch64.AllocationTagFromAddress(address);
if writeback then
    if n == 31 then
        address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;
    else
        X[n] = address;
```
STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

No offset
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
| 1 | x | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | (1) | (1) | (1) | (1) | 0 | (1) | (1) | (1) | (1) | Rn |   |   |   |   |   |   |

| size | L | Rs | o0 | Rt2 |

32-bit (size == 10)

STLLR <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STLLR <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, dbytes, AccType_LIMITEDORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

No offset
(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 1 0 0 0 1 0 0 0 (1) (1) (1) (1) 0 (1) (1) (1) (1) | Rn | 0 0 0 0 1 0 0 0 1 0 0 0 (1) (1) (1) (1) 0 (1) (1) (1) (1) | Rt |

size L Rs o0 Rt2

STLLRB <Wt>, [<Xn|SP>\{,#0}\]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(8) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = X[t];
Mem[address, 1, AccType_LIMITEDORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load LOAcquire, Store LORelease. For information about memory accesses, see Load/Store addressing modes.

No offset
(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|---------------------|---------------------|---------------------|
| 0 1 0 0 1 0 0 0 1 0 0 (1) (1) (1) (1) 0 (1) (1) (1) (1) | Rn | Rt |

size       L       Rs       o0       Rt2

STLLRH <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

data = X[t];
  Mem[address, 2, AccType_LIMITEDORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

<table>
<thead>
<tr>
<th>32-bit (size == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STLR &lt;Wt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit (size == 11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STLR &lt;Xt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

integer n = UInt(Rn);
integer t = UInt(Rt);
integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, dbytes, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

```
0 0 0 0 1 0 0 0 | (1)(1)(1)(1)(1)(1) 1 (1)(1)(1)(1)(1) | Rn | Rt
size     L    Rs  o0     Rt2

STLRB <Wt>, [<Xn|SP>{,#0}]
integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, 1, AccType_ORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 1 0 0 1 0 0 1 0 0 1 0 1 | (1) | (1) | (1) | (1) | (1) | (1) | (1) |
| size | L | Rs | o0 | Rt |

STLRH <Wt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

bits(64) address;
bits(16) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
data = X[t];
Mem[address, 2, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(Armv8.4)

32-bit (size == 10)

STLUR <Wt>, [<Xn|SP>], #<simm>

64-bit (size == 11)

STLUR <Xt>, [<Xn|SP>], #<simm>

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer datasize = 8 << scale;
boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_ORDERED] = data;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STLURB**

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*

For information about memory accesses, see *Load/Store addressing modes*.

**Unscaled offset**

(Armv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | imm9| 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

STLURB `<Wt>`, `<Xn|SP>{, #<simm}>`

bits(64) offset = SignExtend(imm9, 64);

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

\[
\begin{align*}
\text{integer } & n = \text{UInt}(Rn); \\
\text{integer } & t = \text{UInt}(Rt); \\
\text{boolean } & \text{tag\_checked} = n \neq 31;
\end{align*}
\]

**Operation**

\[
\begin{align*}
\text{if } & \text{HaveMTEExt}() \text{ then} \\
& \text{SetTagCheckedInstruction}(\text{tag\_checked}); \\
\text{bits(64) address; } \\
\text{bits(8) data; } \\
\text{if } & n == 31 \text{ then} \\
& \text{CheckSPAlignment}(); \\
& \text{address} = \text{SP}[]; \\
\text{else} & \\
& \text{address} = \text{X}[n]; \\
\text{address} & = \text{address} + \text{offset}; \\
\text{data} & = \text{X}[t]; \\
\text{Mem}[\text{address}, 1, \text{AccType\_ORDERED}] & = \text{data};
\end{align*}
\]

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.

The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release.

For information about memory accesses, see Load/Store addressing modes.

Unscaled offset
(Armv8.4)

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     |     |     | 01  | 0   | 0   | 0   | imm9| 0   | 0   | Rn  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |    |

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, AccType_ORDERED] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

32-bit (sz == 0)

STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]

64-bit (sz == 1)

STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);  // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 3;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
        when Constraint_NONE rt unknown = FALSE;    // store original value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == n & n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
            when Constraint_NONE rn unknown = FALSE;    // address is original base
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXP.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:
0 If the operation updates memory.
1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the “Rt” field.
Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

\[
\text{bits}(64) \text{ address;}
\]
\[
\text{bits}(\text{datasize}) \text{ data;}
\]
\[
\text{constant integer } \text{dbytes} = \text{datasize} \text{ DIV 8;}
\]

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
elsif rn_unknown then
    address = \text{bits}(64) \text{ UNKNOWN;}
else
    address = \text{X}[n];

if rt_unknown then
    data = \text{bits}(\text{datasize}) \text{ UNKNOWN;}
else
    \text{bits}(\text{datasize} \text{ DIV 2}) \text{ el1} = \text{X}[t];
    \text{bits}(\text{datasize} \text{ DIV 2}) \text{ el2} = \text{X}[t2];
    \text{data} = \text{if BigEndian(AccType.ORDERED_ATOMIC) then el1:el2 else el2:el1;}
    \text{bit status} = '1';
    // Check whether the Exclusives monitors are set to include the
    // physical memory locations corresponding to virtual address
    // range [address, address+dbytes-1].
    if AArch64.ExclusiveMonitorsPass(address, dbytes) then
        // This atomic write will be rejected if it does not refer
        // to the same physical locations after address translation.
        Mem[address, dbytes, AccType.ORDERED_ATOMIC] = data;
        status = ExclusiveMonitorsStatus();
    \text{X}[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

32-bit (size == 10)

STLXR <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit (size == 11)

STLXR <Ws>, <Xt>, [<Xn|SP>{,#0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrUncert{DataOverlap};
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_NONE rt_unknown = FALSE; // store original value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrUncert{BaseOverlap};
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_NONE rn_unknown = FALSE; // address is original base
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXR.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:
0 If the operation updates memory.
1 If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- \(<Ws>\) is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```plaintext
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. The memory access is atomic. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
size L o0 Rt2

STLXRB <Ws>, <Wt>, [<Xn|SP>{{,#0}}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
        when Constraint_NONE rt_unknown = FALSE; // store original value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    if s == t then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
            when Constraint_NONE rn_unknown = FALSE; // address is original base
            when Constraint_UNDEF UNDEFINED;
            when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STLXRB.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts
If a synchronous Data Abort exception is generated by the execution of this instruction:

• Memory is not updated.
• <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation

```
bits(64) address;
bits(8) data;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];
if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ORDERED_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 1 0 0 0 0 0 0 0 0 | Rs | 1 |(1)(1)(1)(1)(1) | Rn | Rt |

size <Ws>, <Wt>, [<Xn|SP>{,##0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
        when Constraint_NONE rt unknown = FALSE;    // store original value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    endcase;
endif s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
        when Constraint_NONE rn unknown = FALSE;    // address is original base
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    endcase;
endif s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
        when Constraint_NONE rn unknown = FALSE;    // address is original base
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
    endcase;
endif

For information about the CONstrained UNpredictable behavior of this instruction, see *Architectural Constraints on UNpredictable behaviors*, and particularly *STLXRH*.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Aborts and alignment**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

**Operation**

```plaintext
bits(64) address;
bits(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see Load/Store addressing modes. For information about Non-temporal pair instructions, see Load/Store Non-temporal pair.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 0</td>
</tr>
<tr>
<td>opc</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]  

64-bit (opc == 10)

STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]  

// Empty.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
b bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
**Operation**

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data1 = X[t];
data2 = X[t2];
Mem[address, dbytes, AccType_STREAM] = data1;
Mem[address+dbytes, dbytes, AccType_STREAM] = data2;
```

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STP**

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

**Post-index**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | L  | imm7|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| opc|    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

boolean wback = TRUE;
boolean postindex = TRUE;

**Pre-index**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | L  | imm7|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| opc|    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>!
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>!
```

boolean wback = TRUE;
boolean postindex = TRUE;

**Signed offset**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | L  | imm7|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| opc|    |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>{{, #<imm>}}
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>{{, #<imm>}}
```

boolean wback = FALSE;
boolean postindex = FALSE;
For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STP.

### Assembler Symbols

- **<Wt1>** is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- **<Wt2>** is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- **<Xt1>** is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
- **<Xt2>** is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<imm>** for the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
  - For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
  - For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.
  - For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

### Shared Decode

```plaintext
integer n = Uint(Rn);
integer t = Uint(Rt);
integer t2 = Uint(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + Uint(opc<1>);
integer datasize = 8 << scale;
bv32 offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE rt_unknown = FALSE;   // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;  // value stored is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
```

STP
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
if !postindex then
  address = address + offset;
if rt_unknown && t == n then
  data1 = bits(datasize) UNKNOWN;
else
  data1 = X[t];
if rt_unknown && t2 == n then
  data2 = bits(datasize) UNKNOWN;
else
  data2 = X[t2];
Mem[address, dbytes, AccType_NORMAL] = data1;
Mem[address+dbytes, dbytes, AccType_NORMAL] = data2;
if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STR (immediate)**

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

### Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 x | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | imm9| 0   | 1   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**32-bit (size == 10)**

STR <Wt>, [<Xn|SP>],  #<simm>

**64-bit (size == 11)**

STR <Xt>, [<Xn|SP>],  #<simm>

```java
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bias(64) offset = SignExtend(imm9, 64);
```

### Pre-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 x | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | imm9| 1   | 1   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**32-bit (size == 10)**

STR <Wt>, [<Xn|SP>,  #<simm>]

**64-bit (size == 11)**

STR <Xt>, [<Xn|SP>,  #<simm>]

```java
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bias(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 x | 1   | 1   | 1   | 0   | 0   | 0   | 0   | imm12| 0   | 1   | 0   | 0   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**unsigned offset**

```java
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bias(64) offset = SignExtend(imm12, 64);
```
32-bit (size == 10)

STR <Wt>, [<Xn|SP>{{, #<pimm>}}]

64-bit (size == 11)

STR <Xt>, [<Xn|SP>{{, #<pimm>}}]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.
For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

case wback of
  when FALSE then
    n = UInt(Rn);
    t = UInt(Rt);
    datasize = 8 << scale;
    tag_checked = wback || n != 31;
    rt_unknown = FALSE;
    if wback && n == t && n != 31 then
      c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
      assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
      case c of
        when Constraint_NONE then rt_unknown = FALSE; // value stored is original value
        when Constraint_UNKNOWN then rt_unknown = TRUE; // value stored is UNKNOWN
        when Constraint_UNDEF then UNDEFINED;
        when Constraint_NOP then EndOfInstruction();
    end case;
end case;

STR (immediate)
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
b bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size | opc |

32-bit (size == 10)

STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

64-bit (size == 11)

STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;  // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale;

Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STRB (immediate)**

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

### Post-index

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
size  imm9  0 1  Rn  Rt

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
size  imm9  1 1  Rn  Rt

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
size  imm12  Rn  Rt

boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STRB (immediate)*.

### Assembler Symbols

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>` Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
  c = ConstranUnpredictable(Unpredictable_WBOVERLAPST);
  assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_NONE rt_unknown = FALSE; // value stored is original value
    when Constraint_UNKNOWN rt.unknown = TRUE; // value stored is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  address = address + offset;

if rt_unknown then
  data = bits(8) UNKNOWN;
else
  data = X[t];
Mem[address, 1, AccType_NORMAL] = data;

if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

Extended register (option != 011)

\[
\text{STRB} \ <Wt>, \ [<Xn|SP>, \ (<Wm>|<Xm>), \ <extend> \ {<amount>}]\]

Shifted register (option == 011)

\[
\text{STRB} \ <Wt>, \ [<Xn|SP>, \ <Xm>{, \ LSL \ <amount>}]\]

if option<1> == '0' then UNDEFINED; // sub-word index

\[
\text{ExtendType} \ \text{extend_type} = \text{DecodeRegExtend}(\text{option});\]

Assembler Symbols

\(<Wt>\>
Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\>
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<Wm>\>
When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

\(<Xm>\>
When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

\(<extend>\>
Is the index extend specifier, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

\[
\text{integer} \ n = \text{UInt}(\text{Rn}); \\
\text{integer} \ t = \text{UInt}(\text{Rt}); \\
\text{integer} \ m = \text{UInt}(\text{Rm});\]
Operation

bits(64) offset = ExtendReg(m, extend_type, 0);
if HaveMTEExt() then
    SetTagCheckedInstruction(TRUE);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = X[t];
Mem[address, 1, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

### Post-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|
| 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| size | imm9            | Rn | Rt |
|--------------------------|-----------|    |
```

```plaintext
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|
| 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| size | imm9            | Rn | Rt |
|--------------------------|-----------|    |
```

```plaintext
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|
| 0 1 1 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| size | imm12           | Rn | Rt |
|--------------------------|-----------|    |
```

```plaintext
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STRH (immediate).

### Assembler Symbols

- `<Wt>` is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
- `<pimm>` is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as `<pimm>/2`. 
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];
    Mem[address, 2, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
        if n == 31 then
            SP[] = address;
        else
            X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see Load/Store addressing modes.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|--------------------------------------------------|--------------------------------------------------|--------------------------------------------------|
| 0  1  1  1  1  0  0  0  0  0  1  Rm  option  S  1  0  Rn  Rt | size  opc                                        |
```

```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

- `<Wt>`: Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Wm>`: When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<Xm>`: When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
- `<extend>`: Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when `<amount>` is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

- `<amount>`: Is the index shift amount, optional only when `<extend>` is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```
Operation

bits(64) offset = $\text{ExtendReg}(m, \text{extend type, shift})$;
if $\text{HaveMTEExt}()$ then
  $\text{SetTagCheckedInstruction}(\text{TRUE})$;

bits(64) address;
bits(16) data;

if $n == 31$ then
  $\text{CheckSPAlignment}();$
  address = $\text{SP}[]$;
else
  address = $\text{X}[n]$;

address = address + offset;

data = $\text{X}[t]$;
$\text{Mem}[\text{address, 2, AccType\_NORMAL}] = \text{data}$;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSET does not have release semantics.
- STSETL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSET, LDSETA, LDSETAL, LDSETL**. This means:

- The encodings in this description are named to match the encodings of **LDSET, LDSETA, LDSETAL, LDSETL**.
- The description of **LDSET, LDSETA, LDSETAL, LDSETL** gives the operational pseudocode for this instruction.

### Integer (Armv8.1)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | x  | 1  | 1  | 1  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |   |   |   |   |
```

**32-bit LDSET alias (size == 10 && R == 0)**

STSET <Ws>, [<Xn|SP>]

is equivalent to

LDSET <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**32-bit LDSETL alias (size == 10 && R == 1)**

STSETL <Ws>, [<Xn|SP>]

is equivalent to

LDSETL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDSET alias (size == 11 && R == 0)**

STSET <Xs>, [<Xn|SP>]

is equivalent to

LDSET <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

**64-bit LDSETL alias (size == 11 && R == 1)**

STSETL <Xs>, [<Xn|SP>]

is equivalent to

LDSETL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STSETB, STSETLB**

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB does not have release semantics.
- STSETLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSETB, LDSETAB, LDSETALB, LDSETLB**. This means:

- The encodings in this description are named to match the encodings of **LDSETB, LDSETAB, LDSETALB, LDSETLB**.
- The description of **LDSETB, LDSETAB, LDSETALB, LDSETLB** gives the operational pseudocode for this instruction.

### Integer

(ARMv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | R  | 1  |    | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1 |
| size | A | opc | Rn |

**No memory ordering (R == 0)**

STSETB `<Ws>`, `<Xn|SP>`

is equivalent to

LDSETB `<Ws>`, WZR, `<Xn|SP>`

and is always the preferred disassembly.

**Release (R == 1)**

STSETLB `<Ws>`, `<Xn|SP>`

is equivalent to

LDSETLB `<Ws>`, WZR, `<Xn|SP>`

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of **LDSETB, LDSETAB, LDSETALB, LDSETLB** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH does not have release semantics.
- STSETLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

### Integer

(\text{Armv8.1})

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

#### No memory ordering (R == 0)

STSETH <Ws>, [<Xn|SP>] is equivalent to

LDSETH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### Release (R == 1)

STSETLH <Ws>, [<Xn|SP>]

is equivalent to

LDSETLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX does not have release semantics.
- STSMAXL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL. This means:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
<pre><code>| size | A    | Rs  | opc | Rn  | 1  | 1  | 1  | 1  | 1 |
</code></pre>
```

32-bit LDSMAX alias (size == 10 && R == 0)

STSMAX <Ws>, [<Xn|SP>]

is equivalent to

LDSMAX <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

32-bit LDSMAXL alias (size == 10 && R == 1)

STSMAXL <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDSMAX alias (size == 11 && R == 0)

STSMAX <Xs>, [<Xn|SP>]

is equivalent to

LDSMAX <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDSMAXL alias (size == 11 && R == 1)

STSMAXL <Xs>, [<Xn|SP>]

is equivalent to

LDSMAXL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STSMAXB, STSMAXLB**

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- **STSMAXB** does not have release semantics.
- **STSMAXLB** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB**. This means:

- The encodings in this description are named to match the encodings of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB**.
- The description of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB** gives the operational pseudocode for this instruction.

**Integer**

(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|--------------------------------|------------------|------------------|
| size   | A    | opc   | Rn   | Rt   |
| 0 0 1 1 1 0 0 0 0 | R 1 | Rs   | 0 1 0 0 0 0 | 1 1 1 1 1 1 |

**No memory ordering (R == 0)**

STSMAXB <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Release (R == 1)**

STSMAXLB <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

**Assembler Symbols**

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of **LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAXH does not have release semantics.
- STSMAXLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 0  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

size A opc Rn Rt

No memory ordering (R == 0)

STSMAXH <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSMAXLH <Ws>, [<Xn|SP>]

is equivalent to

LDSMAXLH <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMIN does not have release semantics.
- STSMINL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

### Integer
(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **32-bit LDSMIN alias (size == 10 && R == 0)**

STSMIN <Ws>, [Xn|SP]

is equivalent to

LDSMIN <Ws>, WZR, [Xn|SP]

and is always the preferred disassembly.

- **32-bit LDSMINL alias (size == 10 && R == 1)**

STSMINL <Ws>, [Xn|SP]

is equivalent to

LDSMINL <Ws>, WZR, [Xn|SP]

and is always the preferred disassembly.

- **64-bit LDSMIN alias (size == 11 && R == 0)**

STSMIN <Xs>, [Xn|SP]

is equivalent to

LDSMIN <Xs>, XZR, [Xn|SP]

and is always the preferred disassembly.

- **64-bit LDSMINL alias (size == 11 && R == 1)**

STSMINL <Xs>, [Xn|SP]

is equivalent to

LDSMINL <Xs>, XZR, [Xn|SP]

and is always the preferred disassembly.
**Assembler Symbols**

- `<Ws>`: Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xs>`: Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of `LDSMIN, LDSMINA, LDSMINAL, LDSMINL` gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- STSMINB does not have release semantics.
- STSMINLB stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB. This means:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Integer
(Armv8.1)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | R | 1 | Rs | 0 | 1 | 0 | 1 | 0 | 0 | Rn | 1 | 1 | 1 | 1 | 1 |
| size | A | opc | Rt |

No memory ordering (R == 0)

STSMINB <Ws>, [<Xn|SP>]

is equivalent to

LDSMINB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Release (R == 1)

STSMINLB <Ws>, [<Xn|SP>]

is equivalent to

LDSMINLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

Assembler Symbols

- <Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- **STSMINH** does not have release semantics.
- **STSMINLH** stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

### Integer
**(Armv8.1)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>A</th>
<th>opc</th>
<th>Rt</th>
</tr>
</thead>
</table>

#### No memory ordering (R == 0)

**STSMINH <Ws>, [<Xn|SP>]**

is equivalent to

**LDSMINH <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

#### Release (R == 1)

**STSMINLH <Ws>, [<Xn|SP>]**

is equivalent to

**LDSMINLH <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

### Assembler Symbols

- **<Ws>** is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

32-bit (size == 10)

STTR <Wt>, [<Xn|SP>{, #<simm>}]

64-bit (size == 11)

STTR <Xt>, [<Xn|SP>{, #<simm>}]

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '1';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '1';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
  acctype = AccType_UNPRIV;
else
  acctype = AccType_NORMAL;

integer datasize = 8 << scale;
boolean tag_checked = n != 31;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, acctype] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the **Effective value** of PSTATE.UAO is 0 and either:
- The instruction is executed at EL1.
- The instruction is executed at EL2 when the **Effective value** of HCR_EL2.E2H, TGE is \{1, 1\}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see **Load/Store addressing modes**.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**STTRB <Wt>, [<Xn|SP>{, #<simm}>]**

```
bits(64) offset = SignExtend(imm9, 64);
```

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```python
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled()) && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

**Operation**

```python
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 1, acctype] = data;
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the Effective value of HCR_EL2.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see Load/Store addressing modes.

```
size opc
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 0 0 0 0 0 1 0
imm9 Rn Rt
```

STTRH <Wt>, [<Xn|SP>{{, #<simm>}}]

```
bits(64) offset = SignExtend(imm9, 64);
```

**Assembler Symbols**

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<simm>` Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

**Shared Decode**

```c
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1 && !EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11';
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    accetype = AccType_UNPRIV;
else
    accetype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

**Operation**

```c
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
b bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, 2, accetype] = data;
```
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAX does not have release semantics.
- STUMAXL stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL. This means:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

### Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs |    | 0  | 1  | 1  | 0  | 0  | Rn |    | 1  | 1  | 1  | 1  | 1  |    |    |

<table>
<thead>
<tr>
<th>size</th>
<th>A</th>
<th>opc</th>
<th>Rt</th>
</tr>
</thead>
</table>

32-bit LDUMAX alias (size == 10 && R == 0)

STUMAX <Ws>, [<Xn|SP>]

is equivalent to

LDUMAX <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

32-bit LDUMAXL alias (size == 10 && R == 1)

STUMAXL <Ws>, [<Xn|SP>]

is equivalent to

LDUMAXL <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDUMAX alias (size == 11 && R == 0)

STUMAX <Xs>, [<Xn|SP>]

is equivalent to

LDUMAX <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.

64-bit LDUMAXL alias (size == 11 && R == 1)

STUMAXL <Xs>, [<Xn|SP>]

is equivalent to

LDUMAXL <Xs>, XZR, [<Xn|SP>]

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXB does not have release semantics.
- STUMAXLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of *LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB*. This means:

- The encodings in this description are named to match the encodings of *LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB*.
- The description of *LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB* gives the operational pseudocode for this instruction.

### Integer

(Armv8.1)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **size**<br>- **A**<br>- **opc**<br>- **Rt**

#### No memory ordering (R == 0)

STUMAXB `<Ws>`, `<Xn|SP>`

is equivalent to

LDUMAXB `<Ws>`, WZR, `<Xn|SP>`

and is always the preferred disassembly.

#### Release (R == 1)

STUMAXLB `<Ws>`, `<Xn|SP>`

is equivalent to

LDUMAXLB `<Ws>`, WZR, `<Xn|SP>`

and is always the preferred disassembly.

### Assembler Symbols

- `<Ws>` is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of *LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB* gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH does not have release semantics.
- STUMAXLH stores to memory with release semantics, as described in Load-Acquire, Store-Release.

For information about memory accesses see Load/Store addressing modes.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

### Integer
(Armv8.1)

|       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31    | 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0 |
|  0    |  1|  1|  1|  1|  0|  0|  0|  0|  R|  1 |           |   |   | Rs |  0 |  1 |  1 |  0 |  0 |  0 | Rn |  1 |  1 |  1 |  1 |  1 |
| size  |   |   |   |   |   | A |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| opc  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rt    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

### No memory ordering (R == 0)

STUMAXH $<Ws>$, $<Xn|SP>$

is equivalent to

LDUMAXH $<Ws>$, WZR, $<Xn|SP>$

and is always the preferred disassembly.

### Release (R == 1)

STUMAXLH $<Ws>$, $<Xn|SP>$

is equivalent to

LDUMAXLH $<Ws>$, WZR, $<Xn|SP>$

and is always the preferred disassembly.

### Assembler Symbols

- $<Ws>$ is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- $<Xn|SP>$ is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- **STUMIN** does not have release semantics.
- **STUMINL** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDUMIN, LDUMINA, LDUMINAL, LDUMINL**. This means:

- The encodings in this description are named to match the encodings of **LDUMIN, LDUMINA, LDUMINAL, LDUMINL**.
- The description of **LDUMIN, LDUMINA, LDUMINAL, LDUMINL** gives the operational pseudocode for this instruction.

### Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R  | 1  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  | 1  |

#### 32-bit LDUMIN alias (size == 10 && R == 0)

**STUMIN** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDUMIN** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

#### 32-bit LDUMINL alias (size == 10 && R == 1)

**STUMINL** `<Ws>, [<Xn|SP>]`

is equivalent to

**LDUMINL** `<Ws>, WZR, [<Xn|SP>]`

and is always the preferred disassembly.

#### 64-bit LDUMIN alias (size == 11 && R == 0)

**STUMIN** `<Xs>, [<Xn|SP>]`

is equivalent to

**LDUMIN** `<Xs>, XZR, [<Xn|SP>]`

and is always the preferred disassembly.

#### 64-bit LDUMINL alias (size == 11 && R == 1)

**STUMINL** `<Xs>, [<Xn|SP>]`

is equivalent to

**LDUMINL** `<Xs>, XZR, [<Xn|SP>]`

and is always the preferred disassembly.
Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the
contents of the memory location, encoded in the "Rs" field.

<Xs> Is the 64-bit name of the general-purpose register holding the data value to be operated on with the
contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMIN gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- STUMINB does not have release semantics.
- STUMINLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB. This means:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

### Integer (Armv8.1)

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | R   | 1   | Rs  | 0   | 1   | 1   | 1   | 0   | 0   | Rn  | 1   | 1   | 1   | 1   | 1   |

#### No memory ordering (R == 0)

STUMINB <Ws>, [<Xn|SP>]

is equivalent to

LDUMINB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

#### Release (R == 1)

STUMINLB <Ws>, [<Xn|SP>]

is equivalent to

LDUMINLB <Ws>, WZR, [<Xn|SP>]

and is always the preferred disassembly.

### Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

### Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STUMINH, STUMINLH**

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- **STUMINH** does not have release semantics.
- **STUMINLH** stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of **LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH**. This means:

- The encodings in this description are named to match the encodings of **LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH**.
- The description of **LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH** gives the operational pseudocode for this instruction.

**Integer**  
(Armv8.1)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | R | 1  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn | 1  | 1  | 1  | 1  | 1  |
| opc  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**No memory ordering (R == 0)**

**STUMINH <Ws>, [<Xn|SP>]**

is equivalent to

**LDUMINH <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

**Release (R == 1)**

**STUMINLH <Ws>, [<Xn|SP>]**

is equivalent to

**LDUMINLH <Ws>, WZR, [<Xn|SP>]**

and is always the preferred disassembly.

**Assembler Symbols**

- **<Ws>** is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**Operation**

The description of **LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | imm9 | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (size == 10)

STUR <Wt>, [<Xn|SP>{}, #<simm>]

64-bit (size == 11)

STUR <Xt>, [<Xn|SP>{}, #<simm>]

integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer datasize = 8 << scale;
boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see Load/Store addressing modes.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | imm9 |
|    |    |    |    |    |    | imm9 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Rn |
| size |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    | opc |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

STURB \(<Wt>, \[<Xn|SP>|{}, #<simm>\)\]

bits(64) offset = SignExtend\(imm9, 64\);

Assembler Symbols

\(<Wt>\) Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

\(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<simm>\) Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt\(Rn\);
integer t = UInt\(Rt\);

boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction\(tag_checked\);

bits(64) address;
bits(8) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[ ];
else
    address = X[n];
address = address + offset;

data = X[t];
Mem[address, 1, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see Load/Store addressing modes.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| size                      | opc              | imm9            | Rn              | Rt              |

STURH <Wt>, [<Xn|SP>{, #<simm}>]

bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

bits(64) address;
bits(16) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;
data = X[t];
Mem[address, 2, AccType_NORMAL] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STXP**

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses see *Load/Store addressing modes*.

### 32-bit (sz == 0)

**STXP** `<Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]`

### 64-bit (sz == 1)

**STXP** `<Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]`

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
  Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
    when Constraint_NONE rt unknown = FALSE;   // store original value
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
  case c of
    when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
    when Constraint_NONE rn unknown = FALSE;   // address is original base
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly **STXP**.

**Assembler Symbols**

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:
• Memory is not updated.
• <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
• If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
• Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];

if rt_unknown then
  data = bits(datasize) UNKNOWN;
else
  bits(datasize DIV 2) el1 = X[t];
  bits(datasize DIV 2) el2 = X[t2];
  data = if BigEndian(AccType_ATOMIC) then el1:el2 else el2:el1;
  bit status = '1';
  // Check whether the Exclusives monitors are set to include the
  // physical memory locations corresponding to virtual address
  // range [address, address+dbytes-1].
  if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
  X[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See Synchronization and semaphores. For information about memory accesses see Load/Store addressing modes.

32-bit (size == 10)

STXR <Ws>, <Wt>, [<Xn|SP>{{,#0}}]

64-bit (size == 11)

STXR <Ws>, <Xt>, [<Xn|SP>{{,#0}}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);  // ignored by all loads and store-release

integer elsize = 8 <= UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt unknown = TRUE;  // store UNKNOWN value
        when Constraint_NONE rt unknown = FALSE;    // store original value
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();
if s == n & n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn unknown = TRUE;  // address is UNKNOWN
        when Constraint_NONE rn unknown = FALSE;    // address is original base
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly STXR.

Assembler Symbols

<Ws>  Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0   If the operation updates memory.

1   If the operation fails to update memory.

<Xt>  Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt>  Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment
If a synchronous Data Abort exception is generated by the execution of this instruction:
- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:
- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
elsif rn_unknown then
  address = bits(64) UNKNOWN;
else
  address = X[n];
if rt_unknown then
  data = bits(elsize) UNKNOWN;
else
  data = X[t];
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
  // This atomic write will be rejected if it does not refer
  // to the same physical locations after address translation.
  Mem[address, dbytes, AccType_ATOMIC] = data;
  status = ExclusiveMonitorsStatus();
  X[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**STXRB**

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes*.

```plaintext
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**STXRB** `<Ws>, <Wt>, [<Xn|SP>{,#0}]`

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);  // ignored by all loads and store-release

boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstranUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
      when Constraint_NONE rt Unknown = FALSE;  // store original value
      when Constraint_UNDEF UNDEFINED;
      when Constraint_NOP EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstranUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
          when Constraint_UNKNOWN rn Unknown = TRUE;  // address is UNKNOWN
          when Constraint_NONE rn Unknown = FALSE;  // address is original base
          when Constraint_UNDEF UNDEFINED;
          when Constraint_NOP EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly **STXRB**.

**Assembler Symbols**

- `<Ws>` Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the “Rs” field. The value returned is:
  ```plaintext
  0  If the operation updates memory.
  1  If the operation fails to update memory.
  ```

- `<Wt>` Is the 32-bit name of the general-purpose register to be transferred, encoded in the “Rt” field.

- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.

**Aborts**

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- `<Ws>` is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation

\[
\text{bits(64) address; bits(8) data;}
\]

if \text{HaveMTEExt()} then
\text{SetTagCheckedInstruction(tag\_checked);}\
if \text{n == 31} then
\text{CheckSPadding();}
\text{address = SP[0];}
elsif \text{rn\_unknown} then
\text{address = bits(64) UNKNOWN;}
else
\text{address = X[n];}
endif
if \text{rt\_unknown} then
\text{data = bits(8) UNKNOWN;}
else
\text{data = X[t];}
endif
\text{bit status = '1';}
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if \text{AArch64.ExclusiveMonitorsPass(address, 1)} then
// This atomic write will be rejected if it does not refer
// to the same physical locations after address translation.
\text{Mem[address, 1, AccType\_ATOMIC] = data;}
\text{status = ExclusiveMonitorsStatus();}
\text{X[s] = ZeroExtend(status, 32);}

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes*.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rs</th>
<th>[1(1)(1)(1)(1)]</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 0 0</td>
<td>0 0 0</td>
<td>Rsh</td>
<td>o0</td>
<td>Rt2</td>
<td></td>
</tr>
</tbody>
</table>

STXRH <Ws>, <Wt>, [<Xn|SP>{,0}]

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs); // ignored by all loads and store-release

boolean tag_checked = n != 31;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
  Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rt_unknown = TRUE; // store UNKNOWN value
    when Constraint_NONE rt_unknown = FALSE; // store original value
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();
if s == n & n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rn_unknown = TRUE; // address is UNKNOWN
    when Constraint_NONE rn_unknown = FALSE; // address is original base
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0 If the operation updates memory.

1 If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.
Operation

bits(64) address;
bright(16) data;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn Unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n];

if rt Unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t];

bit status = '1';
// Check whether the Exclusives monitors are set to include the physical memory locations corresponding to virtual address range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer to the same physical locations after address translation.
    Mem[address, 2, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
    X[s] = ZeroExtend(status, 32);

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STZ2G

Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset.

**Post-index**

(Armv8.5)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------|-------------|-------------|
| 1   1   0   1   0   0   1   1   1   | imm9 0 1   | Xn          | Xt          |

STZ2G <Xt|SP>, [<Xn|SP>], #<simm>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

**Pre-index**

(Armv8.5)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------|-------------|-------------|
| 1   1   0   1   0   0   1   1   1   | imm9 1 1   | Xn          | Xt          |

STZ2G <Xt|SP>, [<Xn|SP>], #<simm>]

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

**Signed offset**

(Armv8.5)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------|-------------|-------------|
| 1   1   0   1   0   0   1   1   1   | imm9 1 0   | Xn          | Xt          |

STZ2G <Xt|SP>, [<Xn|SP>], #<simm>]

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

**Assembler Symbols**

<Xt|SP> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<simm> Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
STZG

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

**Post-index**

(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | imm9|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 1  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Xn |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Xt |

STZG <Xt|SP>, [<Xn|SP>], #<simm>

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

**Pre-index**

(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | imm9|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 1  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Xn |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Xt |

STZG <Xt|SP>, [<Xn|SP>], #<simm>]

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

**Signed offset**

(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | imm9|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 1  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Xn |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Xt |

STZG <Xt|SP>, [<Xn|SP>{}, #<simm>}

if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;

**Assembler Symbols**

< Xt|SP > Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.

< Xn|SP > Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

< simm > Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.
Operation

bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AArch64.AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
STZGM

Store Tag and Zero Multiple writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID_EL0.BS, and the Allocation Tag written to address A is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

If ID_AA64FPFR1_EL1.MTE != 0b0010, this instruction is UNDEFINED.

Integer
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Xn  | Xt  |

STZGM <Xt>, [Xn|SP]

if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Xt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t];
bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAAlignment();
    address = SP[ ];
else
    address = X[n];

integer size = 4 * (2 ^ (UInt(DCZID_EL0.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
for i = 0 to count-1
    AArch64_MemTag[address, AccType_NORMAL] = tag;
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);
    address = address + TAG_GRANULE;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

32-bit ($sf == 0$)

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {<$amount>}}
```

64-bit ($sf == 1$)

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {<$amount>}}
```

integer $d = UInt(Rd)$;
integer $n = UInt(Rn)$;
integer $m = UInt(Rm)$;
integer $datasize = if sf == '1' then 64 else 32$;
$ExtendType extend_type = DecodeRegExtend(option)$;
integer $shift = UInt(imm3)$;
if $shift > 4$ then UNDEFINED;

Assembler Symbols

- `<Wd|WSP>` is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>` is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>` is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd|SP>` is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>` is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>` is a width specifier, encoded in "option":
  - option <R>
    - 00x W
    - 010 W
    - x11 X
    - 10x W
    - 110 W

- `<m>` is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- `<extend>` is the extension to be applied to the second source operand, encoded in "option":
  - option <extend>
    - 000 UXTB
    - 001 UXTH
    - 010 LSL|UXTW
    - 011 UXTX
    - 100 SXTB
    - 101 SXTH
    - 110 SXTW
    - 111 SXTX

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
  SP[] = result;
else
  X[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
**SUB (immediate)**

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>sh</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm12</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**32-bit (sf == 0)**

SUB `<Wd|WSP>`, `<Wn|WSP>`, #<imm>{, <shift>}

**64-bit (sf == 1)**

SUB `<Xd|SP>`, `<Xn|SP>`, #<imm>{, <shift>}

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

**Assembler Symbols**

- `<Wd|WSP>` Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Wn|WSP>` Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Xd|SP>` Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
- `<Xn|SP>` Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;

operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
  SP[] = result;
else
  X[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    • The values of the data supplied in any of its registers.
    • The values of the NZCV flags.
**SUB (shifted register)**

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias **NEG (shifted register)**.

### 32-bit (sf == 0)

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' & imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

**Assembler Symbols**

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<shift>** Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<amount>** For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  
  For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG (shifted register)</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, ‘1’);

X[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

**Integer**  
(Armv8.5)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>uimm6</th>
<th></th>
<th>uimm4</th>
<th>Xn</th>
<th>Xd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Symbols**

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.

<uimm6> Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.

<uimm4> Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

**Operation**

```plaintext
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);

if !HaveMTEExt() then UNDEFINED;
if n == 31 then SP[] else X[n];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;
if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
  rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
  rtag = '0000';
(result, -) = AddWithCarry(operand1, NOT(offset), '1');
result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);
if d == 31 then
  SP[] = result;
else
  X[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

Integer
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

```
SUBP <Xd>, <Xn|SP>, <Xm|SP>
```

```
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.

<Xm|SP> Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d] = result;
```
**SUBPS**

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias **CMPP**.

**Integer**

*(Armv8.5)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | Xm | 0  | 0  | 0  | 0  | 0  | Xn | 0  | 0  | 0  | 0  | 0  | Xd |

**Assembler Symbols**

- `<Xd>` is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
- `<Xn|SP>` is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
- `<Xm|SP>` is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPP</td>
<td>$S == '1' &amp;&amp; Xd == '11111'$</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);

bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) operand2 = if m == 31 then SP[] else X[m];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;  
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SUBS (extended register)**

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias **CMP (extended register)**.

**32-bit (sf == 0)**

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

**64-bit (sf == 1)**

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

**Assembler Symbols**

- `<Wd>`: Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn|WSP>`: Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn|SP>`: Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.
- `<R>`: Is a width specifier, encoded in "option":
  ```plaintext
<table>
<thead>
<tr>
<th>option</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>W</td>
</tr>
<tr>
<td>010</td>
<td>W</td>
</tr>
<tr>
<td>x11</td>
<td>X</td>
</tr>
<tr>
<td>10x</td>
<td>W</td>
</tr>
<tr>
<td>110</td>
<td>W</td>
</tr>
</tbody>
</table>
  ```
- `<m>`: Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.
- `<extend>`: For the 32-bit variant: is the extension to be applied to the second source operand, encoded in "option":
  ```plaintext
<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>LSL</td>
</tr>
<tr>
<td>011</td>
<td>UXTW</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SCTX</td>
</tr>
</tbody>
</table>
  ```
If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTH</td>
</tr>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>100</td>
<td>SXTB</td>
</tr>
<tr>
<td>101</td>
<td>SXTH</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (extended register)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

### Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift);
bits(4) nzcv;
operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate).

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>sh</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}

64-bit (sf == 1)

SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #12</td>
</tr>
</tbody>
</table>

case sh of
  when '0' imm = ZeroExtend(imm12, datasize);
  when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);

Assembler Symbols

-Wd- Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
-Wn|WSP- Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
-Xd- Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
-Xn|SP- Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
-<imm>- Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
-<shift>- Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP (immediate)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

Operation

bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[] else X[n];
bits(datasize) operand2;
bits(4) nzcv;
operand2 = NOT(imm);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');
PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUBS (shifted register)**

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases **CMP (shifted register)**, and **NEGS**.

### 32-bit (sf == 0)

```plaintext
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

### 64-bit (sf == 1)

```plaintext
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

### Assembler Symbols

- **<Wd>** is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Wm>** is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<Xd>** is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xn>** is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- **<Xm>** is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- **<shift>** is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<amount>** For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMP (shifted register)</strong></td>
<td>Rd == '11111'</td>
</tr>
<tr>
<td><strong>NEGS</strong></td>
<td>Rn == '11111' &amp; Rd != '11111'</td>
</tr>
</tbody>
</table>
Operation

```c
bits(datasize) result;
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SVC

Supervisor Call causes an exception to be taken to EL1.
On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in ESR_ELx, using the EC value 0x15, and the value of the immediate argument.

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
+--------------------------------------------------+
| 1  1  0  1  0  0  0  0 | imm16 | 0  0  0  0  1 |
```

SVC #<imm>

// Empty.

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.CheckForSVCTrap(imm16);
AArch64.CallSupervisor(imm16);
```
SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, SWPA and SWPAL load from memory with acquire semantics.
- SWPL and SWPAL store to memory with release semantics.
- SWP has neither acquire nor release semantics.

For more information about memory ordering semantics see Load-Acquire, Store-Release. For information about memory accesses see Load/Store addressing modes.

Integer
(Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 1  | 0  | 0  | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

size
32-bit SWP (size == 10 & A == 0 & R == 0)

SWP <Ws>, <Wt>, [<Xn|SP>]

32-bit SWPA (size == 10 & A == 1 & R == 0)

SWPA <Ws>, <Wt>, [<Xn|SP>]

32-bit SWPAL (size == 10 & A == 1 & R == 1)

SWPAL <Ws>, <Wt>, [<Xn|SP>]

32-bit SWPL (size == 10 & A == 0 & R == 1)

SWPL <Ws>, <Wt>, [<Xn|SP>]

64-bit SWP (size == 11 & A == 0 & R == 0)

SWP <Xs>, <Xt>, [<Xn|SP>]

64-bit SWPA (size == 11 & A == 1 & R == 0)

SWPA <Xs>, <Xt>, [<Xn|SP>]

64-bit SWPAL (size == 11 & A == 1 & R == 1)

SWPAL <Xs>, <Xt>, [<Xn|SP>]

64-bit SWPL (size == 11 & A == 0 & R == 1)

SWPL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;

AccType ldacctype = if A == '1' & Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Wt> Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

<Xt> Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, regsize);
### SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has neither acquire nor release semantics.

For more information about memory ordering semantics see [Load-Acquire, Store-Release](#).

For information about memory accesses see [Load/Store addressing modes](#).

#### Integer

**Armv8.1**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 0  | 0  | A  | R  | 1  | Rs | 1  | 0  | 0  | 0  | 0  | 0  | Rn |  |   |   |   |   |   |   |   |   |   |   |   |

**size**

#### SWPB (A == 1 && R == 0)

```c
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPAB (A == 1 && R == 0)

```c
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPALB (A == 1 && R == 1)

```c
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPB (A == 0 && R == 0)

```c
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

#### SWPLB (A == 0 && R == 1)

```c
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

if !HaveAtomicExt() then UNDEFINED;

```c
integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType statcctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

#### Assembler Symbols

- `<Ws>` Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- `<Wt>` Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(8) data;
bits(8) store_value;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSParитет();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);
SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release.*
For information about memory accesses see *Load/Store addressing modes.*

### Integer (Armv8.1)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | A | R | 1 | Rs | 1 | 0 | 0 | 0 | 0 | 0 | Rn | | Rt | | | | | | | |

**size**

### SWPAH (A == 1 && R == 0)

SwPAH <Ws>, <Wt>, [<Xn|SP>]

### SWPALH (A == 1 && R == 1)

SwPALH <Ws>, <Wt>, [<Xn|SP>]

### SWPH (A == 0 && R == 0)

SwPH <Ws>, <Wt>, [<Xn|SP>]

### SWPLH (A == 0 && R == 1)

SwPLH <Ws>, <Wt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;

### Assembler Symbols

- **<Ws>** Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
- **<Wt>** Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Operation

bits(64) address;
bits(16) data;
bits(16) store_value;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

store_value = X[s];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t] = ZeroExtend(data, 32);
SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 && N == 0)

SXTB <Wd>, <Wn>

is equivalent to

SBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

SXTB <Xd>, <Wn>

is equivalent to

SBFM <Xd>, <Xn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of **SBFM**. This means:

- The encodings in this description are named to match the encodings of **SBFM**.
- The description of **SBFM** gives the operational pseudocode for this instruction.

### 32-bit (sf == 0 & N == 0)

**SXTH** `<Wd>, <Wn>`

is equivalent to

**SBFM** `<Wd>, <Wn>, #0, #15`

and is always the preferred disassembly.

### 64-bit (sf == 1 & N == 1)

**SXTH** `<Xd>, <Wn>`

is equivalent to

**SBFM** `<Xd>, <Xn>, #0, #15`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- `<Wn>` Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of **SBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccc}
\hline
\text{sf} & \text{opc} & N & \text{immr} & \text{imms} & \text{Rn} & \text{Rd} \\
\end{array}
\]

64-bit

\text{SXTW} \lll \text{Xd}, \lll \text{Wn} \\

is equivalent to

\text{SBFM} \lll \text{Xd}, \lll \text{Xn}, \lll \#0, \lll \#31

and is always the preferred disassembly.

Assembler Symbols

\text{Xd} \quad \text{Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.}
\text{Xn} \quad \text{Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.}
\text{Wn} \quad \text{Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.}

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SYS**

System instruction. For more information, see `Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions` for the encodings of System instructions.

This instruction is used by the aliases `AT, CFP, CPP, DC, DVP, IC,` and `TLBI`.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|------------------|-----------------|-----------------|------------------|
| 1 1 0 1 0 1 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 | op1  | CRn  | CRm  | op2  | Rt  |

SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

```java
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);
```

`integer t = UInt(Rt);`

`integer sys_op1 = UInt(op1);`

`integer sys_op2 = UInt(op2);`

`integer sys_crn = UInt(CRn);`

`integer sys_crm = UInt(CRm);`

**Assembler Symbols**

**<op1>** Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

**<Cn>** Is a name ‘Cn’, with ‘n’ in the range 0 to 15, encoded in the “CRn” field.

**<Cm>** Is a name ‘Cm’, with ‘m’ in the range 0 to 15, encoded in the "CRm" field.

**<op2>** Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

**<Xt>** Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the "Rt" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>CRn == '0111' &amp;&amp; CRm == '100x' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_AT</td>
</tr>
<tr>
<td>CFP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp; op2 == '100'</td>
</tr>
<tr>
<td>CPP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp; op2 == '111'</td>
</tr>
<tr>
<td>DC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_DC</td>
</tr>
<tr>
<td>DVP</td>
<td>op1 == '011' &amp;&amp; CRn == '0111' &amp;&amp; CRm == '0011' &amp; op2 == '101'</td>
</tr>
<tr>
<td>IC</td>
<td>CRn == '0111' &amp;&amp; SysOp(op1,'0111',CRm,op2) == Sys_IC</td>
</tr>
<tr>
<td>TLBI</td>
<td>CRn == '1000' &amp;&amp; SysOp(op1,'1000',CRm,op2) == Sys_TLBI</td>
</tr>
</tbody>
</table>

**Operation**

```java
AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**System instruction with result. For more information, see Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions** for the encodings of System instructions.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>

`AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);`

```c
integer t = UInt(Rt);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys.crm = UInt(CRm);
```

**Assembler Symbols**

- `<Xt>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
- `<op1>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- `<Cn>` Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
- `<Cm>` Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- `<op2>` Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

**Operation**

// No architecturally defined instructions here.
`X[t] = AArch64.SysInstrWithResult(1, sys_op1, sys_crn, syscrm, sys_op2);`
Programmable table lookup in one or two vector table (zeroing).

Reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements consisting of one or two consecutive vector registers, where the first or only vector holds the lower numbered elements, and places the indexed table element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then it places zero in the corresponding destination vector element.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic.

It has encodings from 2 classes: SVE and SVE2

SVE

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 1 size | Zm | 0 0 1 1 0 0 | Zn | Zd |

TBL <Zd>..<T>, { <Zn>..<T> }, <Zm>..<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean double_table = FALSE;

SVE2

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 1 size | Zm | 0 0 1 0 1 0 | Zn | Zd |

TBL <Zd>..<T>, { <Zn1>..<T>, <Zn2>..<T> }, <Zm>..<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean double_table = TRUE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) indexes = Z[m];
bits(VL) result;
integer table_size = if double_table then VL*2 else VL;
integer table elems = table_size DIV esize;
bits(table_size) table;

if double_table then
    bits(VL) top = Z[(n + 1) MOD 32];
    bits(VL) bottom = Z[n];
    table = (top:bottom)<table_size-1:0>;
else
    table = Z[n];

for e = 0 to elements-1
    integer idx = UInt(Elem[indexes, e, esize]);
    Elem[result, e, esize] = if idx < table elems then Elem[table, idx, esize] else Zeros();
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| b5 | 0  | 1  | 1  | 0  | 1  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

TBNZ <R><t>, #<imm>, <label>

integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);

Assembler Symbols

<op> Is a width specifier, encoded in “b5”:

<table>
<thead>
<tr>
<th>b5</th>
<th>&lt;op&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

bits(datasize) operand = X[t];

if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_DIR);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.

<table>
<thead>
<tr>
<th>b5</th>
<th>b40</th>
<th>imm14</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

TBZ <R><t>, #<imm>, <label>

integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);

Assembler Symbols

<table>
<thead>
<tr>
<th></th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>b5</td>
<td>W</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

bits(datasize) operand = X[t];

if operand<bit_pos> == op then
    BranchTo( PC[ ] + offset, BranchType_DIR);
TCANCEL

This instruction exits Transactional state and discards all state modifications that were performed transactionally. Execution continues at the instruction that follows the TSTART instruction of the outer transaction. The destination register of the TSTART instruction of the outer transaction is written with the immediate operand of TCANCEL.

System
(FEAT_TME)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

TCANCEL #<imm>

if !HaveTME() then UNDEFINED;
boolean retry = (imm16<15> == '1');
bits(15) reason = imm16<14:0>;

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

CheckTMEEnabled();

if TSTATE.depth > 0 then
   FailTransaction(TMFailure_CNCL, retry, FALSE, reason);
TCOMMIT

This instruction commits the current transaction. If the current transaction is an outer transaction, then Transactional state is exited, and all state modifications performed transactionally are committed to the architectural state. TCOMMIT takes no inputs and returns no value. Execution of TCOMMIT is UNDEFINED in Non-transactional state.

System (FEAT_TME)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

TCOMMIT

if !HaveTME() then UNDEFINED;

Operation

CheckTMEEnabled();

if TSTATE.depth == 0 then
  UNDEFINED;

if TSTATE.depth == 1 then
  CommitTransactionalWrites();
  ClearExclusiveLocal(ProcessorID());

TSTATE.depth = TSTATE.depth - 1;
TLBI

TLB Invalidate operation. For more information, see $op0==0b01$, cache maintenance, TLB maintenance, and address translation instructions.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  1 1 0 1 0 1 0 0 | 0 0 1 | op1 | 1 0 0 0 | CRm | op2 | Rt

L
CRn
```

TLBI <tlbi_op>{, <Xt>}

is equivalent to

SYS #<op1>, C8, <Cm>, #<op2>{, <Xt>}

and is the preferred disassembly when $SysOp(op1,'1000',CRm,op2) \text{ \textasciitilde} Sys_{\text{TLBI}}$.

**Assembler Symbols**

- <op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- <Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
- <op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
- <tlbi_op> Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in "op1:CRm:op2":

<table>
<thead>
<tr>
<th>op1</th>
<th>CRm</th>
<th>op2</th>
<th>&lt;tbi_op&gt;</th>
<th>Architectural Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0001</td>
<td>000</td>
<td>VMALLE1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>001</td>
<td>VAE1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>010</td>
<td>ASIDE1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>011</td>
<td>VAAE1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>000</td>
<td>0001</td>
<td>101</td>
<td>VALE1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>001</td>
<td>RVAE1IS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>011</td>
<td>RVAAE1IS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>101</td>
<td>RVALE1IS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0010</td>
<td>111</td>
<td>RVAALE1IS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>000</td>
<td>VMALLE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>001</td>
<td>VAE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>010</td>
<td>ASIDE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>011</td>
<td>VAAE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>101</td>
<td>VALE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0011</td>
<td>111</td>
<td>VAALE1IS</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>001</td>
<td>RVAE1OS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>011</td>
<td>RVAAE1OS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0101</td>
<td>111</td>
<td>RVAALE1OS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>001</td>
<td>RVAE1</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>011</td>
<td>RVAAE1</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>101</td>
<td>RVALE1</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0110</td>
<td>111</td>
<td>RVAALE1</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>000</td>
<td>VMALLE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>001</td>
<td>VAE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>010</td>
<td>ASIDE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>011</td>
<td>VAAE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>101</td>
<td>VALE1</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>0111</td>
<td>111</td>
<td>VAALE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>001</td>
<td>IPAS2EIIS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>010</td>
<td>RIPAS2EIIS</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>101</td>
<td>IPAS2LE1IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0000</td>
<td>110</td>
<td>RIPAS2LE1IS</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>000</td>
<td>ALLE2OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>001</td>
<td>VAE2OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>100</td>
<td>ALLE1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>101</td>
<td>VALE2OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>100</td>
<td>0001</td>
<td>110</td>
<td>VMALLS12E1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>100</td>
<td>0010</td>
<td>001</td>
<td>RVAE2IS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>100</td>
<td>0010</td>
<td>101</td>
<td>RVALE2IS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>000</td>
<td>ALLE2IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>001</td>
<td>VAE2IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>100</td>
<td>ALLE1IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>101</td>
<td>VALE2IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0011</td>
<td>110</td>
<td>VMALLS12E1IS</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>000</td>
<td>IPAS2E1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>001</td>
<td>IPAS2E1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>010</td>
<td>RIPAS2E1</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>011</td>
<td>RIPAS2E1OS</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>100</td>
<td>IPAS2LE1OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>101</td>
<td>IPAS2LE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>110</td>
<td>RIPAS2LE1</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>111</td>
<td>RIPAS2LE1OS</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>100</td>
<td>0101</td>
<td>001</td>
<td>RVAE2OS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>100</td>
<td>0101</td>
<td>101</td>
<td>RVALE2OS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>100</td>
<td>0110</td>
<td>001</td>
<td>RVAE2</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>100</td>
<td>0110</td>
<td>101</td>
<td>RVALE2</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>000</td>
<td>ALLE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>001</td>
<td>VAE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>100</td>
<td>ALLE1</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>101</td>
<td>VALE2</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0111</td>
<td>110</td>
<td>VMALLS12E1</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>000</td>
<td>ALLE3OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>001</td>
<td>VAE3OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>110</td>
<td>0001</td>
<td>101</td>
<td>VALE3OS</td>
<td>FEAT_TLBios</td>
</tr>
<tr>
<td>110</td>
<td>0010</td>
<td>001</td>
<td>RVAE3IS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>110</td>
<td>0010</td>
<td>101</td>
<td>RVALE3IS</td>
<td>FEAT_TLBIRange</td>
</tr>
<tr>
<td>op1</td>
<td>CRm</td>
<td>op2</td>
<td>&lt;tlbi_op&gt;</td>
<td>Architectural Feature</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>----------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>000</td>
<td>ALLE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>001</td>
<td>VAE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0011</td>
<td>101</td>
<td>VALE3IS</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0101</td>
<td>001</td>
<td>RVAE3OS</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0101</td>
<td>101</td>
<td>RVALE3OS</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>001</td>
<td>RVAE3</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0110</td>
<td>101</td>
<td>RVALE3</td>
<td>FEAT_TLBIRANGE</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>000</td>
<td>ALLE3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>001</td>
<td>VAE3</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>0111</td>
<td>101</td>
<td>VALE3</td>
<td>-</td>
</tr>
</tbody>
</table>

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to ‘11111’, encoded in the "Rt" field.

**Operation**

The description of SYS gives the operational pseudocode for this instruction.
TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If FEAT_TRF is not implemented, this instruction executes as a NOP.

System
(Armv8.4)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  |

TSB CSYNC

if !HaveSelfHostedTrace() then EndOfInstruction();

Operation

TraceSynchronizationBarrier();

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**TST (immediate)**

Test bits (immediate), setting the condition flags and discarding the result: \( Rn \ AND \ imm \).

This is an alias of **ANDS (immediate)**. This means:

- The encodings in this description are named to match the encodings of **ANDS (immediate)**.
- The description of **ANDS (immediate)** gives the operational pseudocode for this instruction.

```
+---------+-----+------+
| sf      | N   | immr |
+---------+-----+------+
```

32-bit (\( sf == 0 \) \&\& \( N == 0 \))

TST \(<Wn>, #<imm>\)

is equivalent to

**ANDS** WZR, \(<Wn>, #<imm>\)

and is always the preferred disassembly.

64-bit (\( sf == 1 \))

TST \(<Xn>, #<imm>\)

is equivalent to

**ANDS** XZR, \(<Xn>, #<imm>\)

and is always the preferred disassembly.

**Assembler Symbols**

- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<imm>\) For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr".
  For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

**Operation**

The description of **ANDS (immediate)** gives the operational pseudocode for this instruction.
TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ANDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

### 32-bit (sf == 0)

TST \(<Wn>, <Wm>{, <shift> #<amount>}\)

is equivalent to

ANDS WZR, \(<Wn>, <Wm>{, <shift> #<amount>}\)

and is always the preferred disassembly.

### 64-bit (sf == 1)

TST \(<Xn>, <Xm>{, <shift> #<amount>}\)

is equivalent to

ANDS XZR, \(<Xn>, <Xm>{, <shift> #<amount>}\)

and is always the preferred disassembly.

### Assembler Symbols

- \(<Wn>\) is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Wm>\) is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<Xn>\) is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- \(<Xm>\) is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
- \(<shift>\) is the optional shift to be applied to the final source, defaulting to LSL and encoded in "shift":

<table>
<thead>
<tr>
<th>shift</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LSL</td>
</tr>
<tr>
<td>01</td>
<td>LSR</td>
</tr>
<tr>
<td>10</td>
<td>ASR</td>
</tr>
<tr>
<td>11</td>
<td>ROR</td>
</tr>
</tbody>
</table>

- \(<amount>\) For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
  - For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

### Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
TSTART

This instruction starts a new transaction. If the transaction started successfully, the destination register is set to zero. If the transaction failed or was canceled, then all state modifications that were performed transactionally are discarded and the destination register is written with a non-zero value that encodes the cause of the failure.

System
(FEAT_TME)

1 1 0 1 0 1 0 1 0 0 1 0 0 0 1 1 0 0 1 1 0 0 0 0 0 1 1 | Rt

TSTART <Xt>

if !HaveTME() then UNDEFINED;
integer t = UInt(Rt);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
Operation

CheckTMEEnabled();

boolean IsEL1Regime;
case PSTATE.ELOf
  when EL0
    IsEL1Regime = S1TranslationRegime() == EL1;
    if IsEL1Regime then
      tme = SCTLR_EL1.TME0;
      tmt = SCTLR_EL1.TMT0;
    else
      tme = SCTLR_EL2.TME0;
      tmt = SCTLR_EL2.TMT0;
  when EL1
    tme = SCTLR_EL1.TME;
    tmt = SCTLR_EL1.TMT;
  when EL2
    tme = SCTLR_EL2.TME;
    tmt = SCTLR_EL2.TMT;
  when EL3
    tme = SCTLR_EL3.TME;
    tmt = SCTLR_EL3.TMT;
  otherwise
    Unreachable();

enable = tme == '1';
trivial = tmt == '1';

if !enable then
  TransactionStartTrap(t);
elsif trivial then
  TSTATE.nPC = NextInstrAddr();
  TSTATE.Rt = t;
  FailTransaction(TMFailure_TRIVIAL, FALSE);
elsif TSTATE.depth == 255 then
  FailTransaction(TMFailure_NEST, FALSE);
elsif TSTATE.depth == 0 then
  TSTATE.nPC = NextInstrAddr();
  TSTATE.Rt = t;
  ClearExclusiveLocal(ProcessorID());
  TakeTransactionCheckpoint();
  StartTrackingTransactionalReadsWrites();
else
  TSTATE.depth = TSTATE.depth + 1;
  X[t] = Zeros(64);
This instruction writes the depth of the transaction to the destination register, or the value 0 otherwise.

**System**

**(FEAT_TME)**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rt

TTEST `<Xt>`

if !HaveTME() then UNDEFINED;
integer t = UInt(Rt);

**Assembler Symbols**

`<Xt>`  
Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

**Operation**

`CheckTMEEnabled();`

`X[t] = (TSTATE.depth)<63:0>;`

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UBFIZ

Unsigned Bitfield Insert in Zeros copies a bitfield of \(<width>\) bits from the least significant bits of the source register to bit position \(<lsb>\) of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 1 0 1 0 0 1 1 0 N | immr | imms | Rn | Rd
opc
```

32-bit (sf == 0 && N == 0)

UBFIZ \(<Wd>, <Wn>, #<lsb>, #<width>\)

is equivalent to

UBFM \(<Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)\)

and is the preferred disassembly when UInt(imms) < UInt(immr).

64-bit (sf == 1 && N == 1)

UBFIZ \(<Xd>, <Xn>, #<lsb>, #<width>\)

is equivalent to

UBFM \(<Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)\)

and is the preferred disassembly when UInt(imms) < UInt(immr).

**Assembler Symbols**

- \(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<lsb>\) For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
- \(<width>\) For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

**Operation**

The description of UBFM gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UBFM

Unsigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly. If \( \text{immr} \) is greater than or equal to \( \text{imms} \), this copies a bitfield of \( (\text{imms} - \text{immr} + 1) \) bits starting from bit position \( \text{immr} \) in the source register to the least significant bits of the destination register. If \( \text{imms} \) is less than \( \text{immr} \), this copies a bitfield of \( (\text{imms} + 1) \) bits from the least significant bits of the source register to bit position \( \text{regsize} - \text{immr} \) of the destination register, where \( \text{regsize} \) is the destination register size of 32 or 64 bits. In both cases the destination bits above and below the bitfield are set to zero.

This instruction is used by the aliases \text{LSL (immediate)}, \text{LSR (immediate)}, \text{UBFIZ}, \text{UBFX}, \text{UXTB}, and \text{UXTH}.

32-bit (\( \text{sf} = 0 \&\& \text{N} = 0 \))

\[
\text{UBFM} \ <Wd>, <Wn>, \ #\text{immr}, \ #\text{imms} 
\]

64-bit (\( \text{sf} = 1 \&\& \text{N} = 1 \))

\[
\text{UBFM} \ <Xd>, <Xn>, \ #\text{immr}, \ #\text{imms} 
\]

Assembler Symbols

\(<Wd>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Wn>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

\(<Xd>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

\(<Xn>\) Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

\(<\text{immr}>\) For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.

\(<\text{imms}>\) For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{LSL (immediate)}</td>
<td>32-bit</td>
<td>\text{imms} \neq '011111' &amp;&amp; \text{imms} + 1 \neq \text{immr}</td>
</tr>
<tr>
<td>\text{LSL (immediate)}</td>
<td>64-bit</td>
<td>\text{imms} \neq '111111' &amp;&amp; \text{imms} + 1 \neq \text{immr}</td>
</tr>
<tr>
<td>\text{LSR (immediate)}</td>
<td>32-bit</td>
<td>\text{imms} = '011111'</td>
</tr>
</tbody>
</table>
### Alias

<table>
<thead>
<tr>
<th>Alias</th>
<th>Of variant</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSR (immediate)</td>
<td>64-bit</td>
<td><code>imms == '111111'</code></td>
</tr>
<tr>
<td>UBFIZ</td>
<td></td>
<td><code>UInt(imms) &lt; UInt(immr)</code></td>
</tr>
<tr>
<td>UBFX</td>
<td></td>
<td><code>BFXPreferred(sf, opc&lt;1&gt;, imms, immr)</code></td>
</tr>
<tr>
<td>UXTB</td>
<td></td>
<td><code>immr == '000000' &amp;&amp; imms == '000111'</code></td>
</tr>
<tr>
<td>UXTH</td>
<td></td>
<td><code>immr == '000000' &amp;&amp; imms == '001111'</code></td>
</tr>
</tbody>
</table>

### Operation

```c
bits(datasize) src = X[n];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, R) AND wmask;

// combine extension bits and result bits
X[d] = bot AND tmask;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UBFX

Unsigned Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

32-bit (sf == 0 && N == 0)

UBFX <Wd>, <Wn>, #<lsb>, #<width>

is equivalent to

UBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

64-bit (sf == 1 && N == 1)

UBFX <Xd>, <Xn>, #<lsb>, #<width>

is equivalent to

UBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)

and is the preferred disassembly when BFXPreferred(sf, opc<1>, imms, immr).

Assembler Symbols

- <Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- <Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- <Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
- <lsb> For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31.
  For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
- <width> For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>.
  For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanently UNDEFINED in the Armv8-A architecture.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | imm16 |

UDF #<imm>

// The imm16 field is ignored by hardware.
UNDEFINED;

Assembler Symbols

<imm> is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

Operation

// No operation.
### UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit (sf == 0)

```
UDIV <Wd>, <Wn>, <Wm>
```

#### 64-bit (sf == 1)

```
UDIV <Xd>, <Xn>, <Xm>
```

```plaintext
integer d = Uint(Rd);
n = Uint(Rn);
m = Uint(Rm);
datasize = if sf == '1' then 64 else 32;
```

**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Wm>` Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xn>` Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

#### Operation

```plaintext
bits(datasize) operand1 = X[n];
bits(datasize) operand2 = X[m];
integer result;
if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));
X[d] = result<datasize-1:0>;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias **UMULL**.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  |

UMADDL `<Xd>`, `<Wn>`, `<Wm>`, `<Xa>`

integer `d = UInt(Rd);`
integer `n = UInt(Rn);`
integer `m = UInt(Rm);`
integer `a = UInt(Ra);`

**Assembler Symbols**

`<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

`<Wn>` Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

`<Wm>` Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

`<Xa>` Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UMULL</strong></td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```
bits(32) operand1 = X[n];
bits(32) operand2 = X[m];
bits(64) operand3 = X[a];

integer result;
result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));
X[d] = result<63:0>;
```
UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values,negates the product, and writes the result to the 64-bit destination register.

This is an alias of UMSUBL. This means:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | Rm | 1  | 1  | 1  | 1  | 1  | Rn | 1  |   |   |   |   |   |   |   |   |   |

UMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

UMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMSUBL**

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias **UMNEGL**.

![32-bit register values](image)

**Assembler Symbols**

- `<Xd>`: Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>`: Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- `<Wm>`: Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
- `<Xa>`: Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMNEGL</td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

d = Uint(Rd);
n = Uint(Rn);
m = Uint(Rm);
a = Uint(Ra);

result = Int(operand3, TRUE) - (Int(operand1, TRUE) * Int(operand2, TRUE));
Xd = result<63:0>;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>(1)(1)(1)(1)(1)</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UMULH <Xd>, <Xn>, <Xm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

bits(64) operand1 = X[n];
bits(64) operand2 = X[m];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d] = result<127:64>;

Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of **UMADDL**. This means:

- The encodings in this description are named to match the encodings of **UMADDL**.
- The description of **UMADDL** gives the operational pseudocode for this instruction.

![Register Encoding Diagram]

**UMULL** \(<Xd>, <Wn>, <Wm>\)

is equivalent to

**UMADDL** \(<Xd>, <Wn>, <Wm>, XZR\)

and is always the preferred disassembly.

**Assembler Symbols**

- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Wn>** Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- **<Wm>** Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

**Operation**

The description of **UMADDL** gives the operational pseudocode for this instruction.

**Operational information**

If **PSTATE.DIT** is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

---

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sf  opc  N  immr  imms  Rd

32-bit

UXTB <Wd>, <Wn>

is equivalent to

UBFM <Wd>, <Wn>, #0, #7

and is always the preferred disassembly.

Assembler Symbols

<Wd>  Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>  Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of **UBFM**. This means:

- The encodings in this description are named to match the encodings of **UBFM**.
- The description of **UBFM** gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| sf | opc| N  | immr| imms|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**32-bit**

UXTH `<Wd>`, `<Wn>`

is equivalent to

**UBFM** `<Wd>`, `<Wn>`, #0, #15

and is always the preferred disassembly.

**Assembler Symbols**

- `<Wd>` is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Wn>` is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

**Operation**

The description of **UBFM** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*.

As described in *Wait For Event mechanism and Send event*, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- **Traps to EL1 of EL0 execution of WFE and WFI instructions.**
- **Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.**
- **Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.**

```
0  CRm  | 1 0 1 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 1 1 1 1 1
1  31  30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

WFE

// Empty.

Operation

```
Hint_WFE(-1);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**WFET**

Wait For Event with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*.

As described in *Wait For Event mechanism and Send event*, the execution of a WFET instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- Traps to EL1 of EL0 execution of WFE and WFI instructions.
- Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.
- Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.

---

**System**

(Armv8.7)

```
0 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 | Rd
| op2
```

**WFET** <Xt>

```plaintext
if !HaveFeatWFxT() then UNDEFINED;

integer d = UInt(Rd);
```

**Assembler Symbols**

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

**Operation**

```plaintext
bits(64) operand = X[d];
integer localtimeout = UInt(operand);
if Halted() & constrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();
Hint_WFE(localtimeout);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, swe v2020-09 rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt*.

As described in *Wait For Interrupt*, the execution of a WFI instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- *Traps to EL1 of EL0 execution of WFE and WFI instructions.*
- *Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.*
- *Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.*

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

WFI

// Empty.

**Operation**

```
Hint_WFI(-1);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
WFIT

Wait For Interrupt with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see Wait For Interrupt. As described in Wait For Interrupt, the execution of a WFIT instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level. See:

- Traps to EL1 of EL0 execution of WFE and WFI instructions.
- Traps to EL2 of Non-secure EL0 and EL1 execution of WFE and WFI instructions.
- Traps to EL3 of EL2, EL1, and EL0 execution of WFE and WFI instructions.

System
(Armv8.7)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | Rd |

WFIT <Xt>

if !HaveFeatWFxT() then UNDEFINED;

integer d = UINT(Rd);

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

bits(64) operand = X[d];
integer localtimeout = UINT(operand);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
  EndOfInstruction();

Hint_WFI(localtimeout);
XAFLAG

Convert floating-point condition flags from external format to Arm format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

System
(Armv8.5)

if !HaveFlagFormatExt() then UNDEFINED;

Operation

bit N = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bzet Z = PSTATE.Z AND PSTATE.C;
bzet C = PSTATE.C OR PSTATE.Z;
bzet V = NOT(PSTATE.C) AND PSTATE.Z;
PSTATE.N = N;
PSTATE.Z = Z;
PSTATE.C = C;
PSTATE.V = V;
XPACD, XPACI, XPACLRI

Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPACI and XPACD, and is in LR for XPACLRI. The XPACD instruction is used for data addresses, and XPACI and XPACLRI are used for instruction addresses.

It has encodings from 2 classes: Integer and System

### Integer
(Armv8.3)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

XPACD (D == 1)

XPACD <Xd>

XPACI (D == 0)

XPACI <Xd>

```java
boolean data = (D == '1');
integer d = UInt(Rd);
if !HavePACExt() then
    UNDEFINED;
```

### System
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0   |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

XPACLRI

```java
integer d = 30;
boolean data = FALSE;
```

### Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

### Operation

```java
if HavePACExt() then
    X[d] = Strip(X[d], data);
```
YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a YIELD instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see The YIELD instruction.

```
// Empty.

// Operation

Hint_Yield();
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
A64 -- SIMD and Floating-point Instructions (alphabetic order)

**ABS**: Absolute value (vector).

**ADD (vector)**: Add (vector).

**ADDHN, ADDHN2**: Add returning High Narrow.

**ADDP (scalar)**: Add Pair of elements (scalar).

**ADDP (vector)**: Add Pairwise (vector).

**ADDV**: Add across Vector.

**AESD**: AES single round decryption.

**AESE**: AES single round encryption.

**AESIMC**: AES inverse mix columns.

**AESMC**: AES mix columns.

**AND (vector)**: Bitwise AND (vector).

**BCAX**: Bit Clear and XOR.

**BFCVT**: Floating-point convert from single-precision to BFloat16 format (scalar).

**BFCVTN, BFCVTN2**: Floating-point convert from single-precision to BFloat16 format (vector).

**BFDOT (by element)**: BFloat16 floating-point dot product (vector, by element).

**BFDOT (vector)**: BFloat16 floating-point dot product (vector).

**BFMLALB, BFMLALT (by element)**: BFloat16 floating-point widening multiply-add long (by element).

**BFMLALB, BFMLALT (vector)**: BFloat16 floating-point widening multiply-add long (vector).

**BFMMLA**: BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.

**BIC (vector, immediate)**: Bitwise bit Clear (vector, immediate).

**BIC (vector, register)**: Bitwise bit Clear (vector, register).

**BIF**: Bitwise Insert if False.

**BIT**: Bitwise Insert if True.

**BSL**: Bitwise Select.

**CLS (vector)**: Count Leading Sign bits (vector).

**CLZ (vector)**: Count Leading Zero bits (vector).

**CMEQ (register)**: Compare bitwise Equal (vector).

**CMEQ (zero)**: Compare bitwise Equal to zero (vector).

**CMGE (register)**: Compare signed Greater than or Equal (vector).

**CMGE (zero)**: Compare signed Greater than or Equal to zero (vector).

**CMGT (register)**: Compare signed Greater than (vector).

**CMGT (zero)**: Compare signed Greater than zero (vector).

**CMHI (register)**: Compare unsigned Higher (vector).

**CMHS (register)**: Compare unsigned Higher or Same (vector).
CMLE (zero): Compare signed Less than or Equal to zero (vector).
CMLT (zero): Compare signed Less than zero (vector).
CMST: Compare bitwise Test bits nonzero (vector).
CNT: Population Count per byte.
DUP (element): Duplicate vector element to vector or scalar.
DUP (general): Duplicate general-purpose register to vector.
EOR (vector): Bitwise Exclusive OR (vector).
EOR3: Three-way Exclusive OR.
EXT: Extract vector from pair of vectors.
FABD: Floating-point Absolute Difference (vector).
FABS (scalar): Floating-point Absolute value (scalar).
FABS (vector): Floating-point Absolute value (vector).
FACGE: Floating-point Absolute Compare Greater than or Equal (vector).
FACGT: Floating-point Absolute Compare Greater than (vector).
FADD (scalar): Floating-point Add (scalar).
FADD (vector): Floating-point Add (vector).
FADDP (scalar): Floating-point Add Pair of elements (scalar).
FADDP (vector): Floating-point Add Pairwise (vector).
FCADD: Floating-point Complex Add.
FCCMP: Floating-point Conditional quiet Compare (scalar).
FCCMPE: Floating-point Conditional signaling Compare (scalar).
FCMEQ (register): Floating-point Compare Equal (vector).
FCMEQ (zero): Floating-point Compare Equal to zero (vector).
FCMGE (register): Floating-point Compare Greater than or Equal (vector).
FCMGE (zero): Floating-point Compare Greater than or Equal to zero (vector).
FCMGTE (register): Floating-point Compare Greater than (vector).
FCMGTE (zero): Floating-point Compare Greater than zero (vector).
FCMLA: Floating-point Complex Multiply Accumulate.
FCMLA (by element): Floating-point Complex Multiply Accumulate (by element).
FCMLE (zero): Floating-point Compare Less than or Equal to zero (vector).
FCMLT (zero): Floating-point Compare Less than zero (vector).
FCMP: Floating-point quiet Compare (scalar).
FCMPE: Floating-point signaling Compare (scalar).
FCSEL: Floating-point Conditional Select (scalar).
FCVT: Floating-point Convert precision (scalar).
FCVTAS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).
FCVTAS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).
FCVTAU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).
FCVTAU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).
FCVTL, FCVTL2: Floating-point Convert to higher precision Long (vector).
FCVTMS (scalar): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).
FCVTMS (vector): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).
FCVTMU (scalar): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).
FCVTMU (vector): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).
FCVTN, FCVTN2: Floating-point Convert to lower precision Narrow (vector).
FCVTNS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).
FCVTNS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).
FCVTNU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).
FCVTNU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).
FCVTPS (scalar): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).
FCVTPS (vector): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).
FCVTPU (scalar): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).
FCVTPU (vector): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).
FCVTXN, FCVTXN2: Floating-point Convert to lower precision Narrow, rounding to odd (vector).
FCVTZS (scalar, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).
FCVTZS (scalar, integer): Floating-point Convert to Signed integer, rounding toward Zero (scalar).
FCVTZS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTZS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
FCVTZU (scalar, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).
FCVTZU (scalar, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).
FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).
FDIV (scalar): Floating-point Divide (scalar).
FDIV (vector): Floating-point Divide (vector).
FCVTZS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.
FMADD: Floating-point fused Multiply-Add (scalar).
FMAX (scalar): Floating-point Maximum (scalar).
FMAX (vector): Floating-point Maximum (vector).
FMAXNM (scalar): Floating-point Maximum Number (scalar).
FMAXNM (vector): Floating-point Maximum Number (vector).
FMAXNMP (scalar): Floating-point Maximum Number of Pair of elements (scalar).
FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).
FMAXNMV: Floating-point Maximum Number across Vector.

FMAXP (scalar): Floating-point Maximum of Pair of elements (scalar).

FMAXP (vector): Floating-point Maximum Pairwise (vector).

FMAXV: Floating-point Maximum across Vector.

FMINP (scalar): Floating-point Minimum of Pair of elements (scalar).

FMINP (vector): Floating-point Minimum Pairwise (vector).

FMINV: Floating-point Minimum across Vector.

FMINNM: Floating-point Minimum Number (scalar).

FMINNM (vector): Floating-point Minimum Number Pairwise (vector).

FMINNMV: Floating-point Minimum Number across Vector.

FMINNMP (scalar): Floating-point Minimum Number of Pair of elements (scalar).

FMINNMP (vector): Floating-point Minimum Number Pairwise (vector).

FMLA (by element): Floating-point fused Multiply-Add to accumulator (by element).

FMLA (vector): Floating-point fused Multiply-Add to accumulator (vector).

FMLAL, FMLAL2 (by element): Floating-point fused Multiply-Add Long to accumulator (by element).

FMLAL, FMLAL2 (vector): Floating-point fused Multiply-Add Long to accumulator (vector).

FLMS (by element): Floating-point fused Multiply-Subtract from accumulator (by element).

FLMS (vector): Floating-point fused Multiply-Subtract from accumulator (vector).

FMLSL, FMLSL2 (by element): Floating-point fused Multiply-Subtract Long from accumulator (by element).

FMLSL, FMLSL2 (vector): Floating-point fused Multiply-Subtract Long from accumulator (vector).

FMOV (general): Floating-point Move to or from general-purpose register without conversion.

FMOV (register): Floating-point Move register without conversion.

FMOV (scalar, immediate): Floating-point move immediate (scalar).

FMOV (vector, immediate): Floating-point move immediate (vector).

FMULX: Floating-point Multiply extended.

FMULX (by element): Floating-point Multiply extended (by element).

FNEG (scalar): Floating-point Negate (scalar).

FNEG (vector): Floating-point Negate (vector).

FNMSUB: Floating-point Negated fused Multiply-Subtract (scalar).
FNMUL (scalar): Floating-point Multiply-Negate (scalar).
FRECPE: Floating-point Reciprocal Estimate.
FRECPS: Floating-point Reciprocal Step.
FRECPSX: Floating-point Reciprocal exponent (scalar).
FRINT32X (scalar): Floating-point Round to 32-bit Integer, using current rounding mode (scalar).
FRINT32X (vector): Floating-point Round to 32-bit Integer, using current rounding mode (vector).
FRINT32Z (scalar): Floating-point Round to 32-bit Integer toward Zero (scalar).
FRINT32Z (vector): Floating-point Round to 32-bit Integer toward Zero (vector).
FRINT64X (scalar): Floating-point Round to 64-bit Integer, using current rounding mode (scalar).
FRINT64X (vector): Floating-point Round to 64-bit Integer, using current rounding mode (vector).
FRINT64Z (scalar): Floating-point Round to 64-bit Integer toward Zero (scalar).
FRINT64Z (vector): Floating-point Round to 64-bit Integer toward Zero (vector).
FRINTA (scalar): Floating-point Round to Integral, to nearest with ties to Away (scalar).
FRINTA (vector): Floating-point Round to Integral, to nearest with ties to Away (vector).
FRINTI (scalar): Floating-point Round to Integral, using current rounding mode (scalar).
FRINTI (vector): Floating-point Round to Integral, using current rounding mode (vector).
FRINTM (scalar): Floating-point Round to Integral, toward Minus infinity (scalar).
FRINTM (vector): Floating-point Round to Integral, toward Minus infinity (vector).
FRINTN (scalar): Floating-point Round to Integral, to nearest with ties to even (scalar).
FRINTN (vector): Floating-point Round to Integral, to nearest with ties to even (vector).
FRINTP (scalar): Floating-point Round to Integral, toward Plus infinity (scalar).
FRINTP (vector): Floating-point Round to Integral, toward Plus infinity (vector).
FRINTX (scalar): Floating-point Round to Integral exact, using current rounding mode (scalar).
FRINTX (vector): Floating-point Round to Integral exact, using current rounding mode (vector).
FRINTZ (scalar): Floating-point Round to Integral, toward Zero (scalar).
FRINTZ (vector): Floating-point Round to Integral, toward Zero (vector).
FRSQRTE: Floating-point Reciprocal Square Root Estimate.
FRSQRTS: Floating-point Reciprocal Square Root Step.
FSORT (scalar): Floating-point Square Root (scalar).
FSORT (vector): Floating-point Square Root (vector).
FSUB (scalar): Floating-point Subtract (scalar).
FSUB (vector): Floating-point Subtract (vector).
INS (element): Insert vector element from another vector element.
INS (general): Insert vector element from general-purpose register.
LD1 (multiple structures): Load multiple single-element structures to one, two, three, or four registers.
LD1 (single structure): Load one single-element structure to one lane of one register.
LD1R: Load one single-element structure and Replicate to all lanes (of one register).
LD2 (multiple structures): Load multiple 2-element structures to two registers.
LD2 (single structure): Load single 2-element structure to one lane of two registers.
LD2R: Load single 2-element structure and Replicate to all lanes of two registers.
LD3 (multiple structures): Load multiple 3-element structures to three registers.
LD3 (single structure): Load single 3-element structure to one lane of three registers.
LD3R: Load single 3-element structure and Replicate to all lanes of three registers.
LD4 (multiple structures): Load multiple 4-element structures to four registers.
LD4 (single structure): Load single 4-element structure to one lane of four registers.
LD4R: Load single 4-element structure and Replicate to all lanes of four registers.
LDNP (SIMD&FP): Load Pair of SIMD&FP registers, with Non-temporal hint.
LDP (SIMD&FP): Load Pair of SIMD&FP registers.
LDR (immediate, SIMD&FP): Load SIMD&FP Register (immediate offset).
LDR (literal, SIMD&FP): Load SIMD&FP Register (PC-relative literal).
LDR (register, SIMD&FP): Load SIMD&FP Register (register offset).
LDUR (SIMD&FP): Load SIMD&FP Register (unscaled offset).
MLA (by element): Multiply-Add to accumulator (vector, by element).
MLA (vector): Multiply-Add to accumulator (vector).
MLS (by element): Multiply-Subtract from accumulator (vector, by element).
MLS (vector): Multiply-Subtract from accumulator (vector).
MOV (element): Move vector element to another vector element: an alias of INS (element).
MOV (from general): Move general-purpose register to a vector element: an alias of INS (general).
MOV (scalar): Move vector element to scalar: an alias of DUP (element).
MOV (to general): Move vector element to general-purpose register: an alias of UMOV.
MOVI: Move Immediate (vector).
MUL (by element): Multiply (vector, by element).
MUL (vector): Multiply (vector).
MVN: Bitwise NOT (vector): an alias of NOT.
MVNI: Move inverted Immediate (vector).
NEG (vector): Negate (vector).
NOT: Bitwise NOT (vector).
ORN (vector): Bitwise inclusive OR NOT (vector).
ORR (vector, immediate): Bitwise inclusive OR (vector, immediate).
ORR (vector, register): Bitwise inclusive OR (vector, register).
PMUL: Polynomial Multiply.
PMULL, PMULL2: Polynomial Multiply Long.
RADDHN, RADDHN2: Rounding Add returning High Narrow.
RAX1: Rotate and Exclusive OR.
RBIT (vector): Reverse Bit order (vector).
REV16 (vector): Reverse elements in 16-bit halfwords (vector).
REV32 (vector): Reverse elements in 32-bit words (vector).
REV64: Reverse elements in 64-bit doublewords (vector).
RSHRN, RSHRN2: Rounding Shift Right Narrow (immediate).
RSUBHN, RSUBHN2: Rounding Subtract returning High Narrow.
SABA: Signed Absolute difference and Accumulate.
SABAL, SABAL2: Signed Absolute difference and Accumulate Long.
SABD: Signed Absolute Difference.
SABDL, SABDL2: Signed Absolute Difference Long.
SADALP: Signed Add and Accumulate Long Pairwise.
SADDL, SADDL2: Signed Add Long (vector).
SADLP: Signed Add Long Pairwise.
SADDLV: Signed Add Long across Vector.
SADDW, SADDW2: Signed Add Wide.
SCVTF (scalar, fixed-point): Signed fixed-point Convert to Floating-point (scalar).
SCVTF (scalar, integer): Signed integer Convert to Floating-point (scalar).
SCVTF (vector, fixed-point): Signed fixed-point Convert to Floating-point (vector).
SCVTF (vector, integer): Signed integer Convert to Floating-point (vector).
SDOT (by element): Dot Product signed arithmetic (vector, by element).
SDOT (vector): Dot Product signed arithmetic (vector).
SHA1C: SHA1 hash update (choose).
SHA1H: SHA1 fixed rotate.
SHA1M: SHA1 hash update (majority).
SHA1P: SHA1 hash update (parity).
SHA1SU0: SHA1 schedule update 0.
SHA1SU1: SHA1 schedule update 1.
SHA256H: SHA256 hash update (part 1).
SHA256H2: SHA256 hash update (part 2).
SHA256SU0: SHA256 schedule update 0.
SHA256SU1: SHA256 schedule update 1.
SHA512H: SHA512 Hash update part 1.
SHA512H2: SHA512 Hash update part 2.
SHA512SU0: SHA512 Schedule Update 0.
SHA512SU1: SHA512 Schedule Update 1.
SHADD: Signed Halving Add.
SHL: Shift Left (immediate).
SHLL, SHLL2: Shift Left Long (by element size).
SHRN, SHRN2: Shift Right Narrow (immediate).
SHSUB: Signed Halving Subtract.
SLI: Shift Left and Insert (immediate).
SM3PARTW1: SM3PARTW1.
SM3PARTW2: SM3PARTW2.
SM3SS1: SM3SS1.
SM3TT1A: SM3TT1A.
SM3TT1B: SM3TT1B.
SM3TT2A: SM3TT2A.
SM3TT2B: SM3TT2B.
SM4E: SM4 Encode.
SM4EKEY: SM4 Key.
SMAX: Signed Maximum (vector).
SMAXP: Signed Maximum Pairwise.
SMAXV: Signed Maximum across Vector.
SMIN: Signed Minimum (vector).
SMINP: Signed Minimum Pairwise.
SMINV: Signed Minimum across Vector.
SMLSL, SMLSL2 (by element): Signed Multiply-Subtract Long (vector, by element).
SMMLA (vector): Signed 8-bit integer matrix multiply-accumulate (vector).
SMOV: Signed Move vector element to general-purpose register.
SQABS: Signed saturating Absolute value.
SQADD: Signed saturating Add.
SQDMLSL, SQDMLSL2 (by element): Signed saturating Doubling Multiply-Subtract Long (by element).

SQDMULH (by element): Signed saturating Doubling Multiply returning High half (by element).

SQDMULH (vector): Signed saturating Doubling Multiply returning High half.

SQDMULL, SQDMULL2 (by element): Signed saturating Doubling Multiply Long (by element).


SQNEG: Signed saturating Negate.

SQDMULH (by element): Signed saturating Doubling Multiply returning High half (by element).

SQDMULH (vector): Signed saturating Doubling Multiply returning High half.

SQDMULL, SQDMULL2 (by element): Signed saturating Doubling Multiply Long (by element).


SQNEG: Signed saturating Negate.

SQRDMLAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

SQRDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

SQRDMLSH (by element): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

SQRDMLSH (vector): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

SORSHL: Signed saturating Rounding Shift Left (register).

SORSHRN, SORSHRN2: Signed saturating Rounding Shift Right Narrow (immediate).

SORSHRUN, SORSHRUN2: Signed saturating Rounding Shift Right Unsigned Narrow (immediate).

SOSHL (immediate): Signed saturating Shift Left (immediate).

SOSHL (register): Signed saturating Shift Left (register).

SOSHLU: Signed saturating Shift Left Unsigned (immediate).

SOSHRN, SOSHRN2: Signed saturating Shift Right Narrow (immediate).

SOSHRUN, SOSHRUN2: Signed saturating Shift Right Unsigned Narrow (immediate).

SQSUB: Signed saturating Subtract.

SOXTN, SOXTN2: Signed saturating extract Narrow.

SOXTUN, SOXTUN2: Signed saturating extract Unsigned Narrow.

SRHADD: Signed Rounding Halving Add.

SRI: Shift Right and Insert (immediate).

SRSHL: Signed Rounding Shift Left (register).

SRSHR: Signed Rounding Shift Right (immediate).

SRSRA: Signed Rounding Shift Right and Accumulate (immediate).

SSHL: Signed Shift Left (register).

SSHL, SSHLL2: Signed Shift Left Long (immediate).

SSHR: Signed Shift Right (immediate).

SSRA: Signed Shift Right and Accumulate (immediate).

SSUBL, SSUBL2: Signed Subtract Long.

SSUBW, SSUBW2: Signed Subtract Wide.

ST1 (multiple structures): Store multiple single-element structures from one, two, three, or four registers.

ST1 (single structure): Store a single-element structure from one lane of one register.
ST2 (multiple structures): Store multiple 2-element structures from two registers.
ST2 (single structure): Store single 2-element structure from one lane of two registers.
ST3 (multiple structures): Store multiple 3-element structures from three registers.
ST3 (single structure): Store single 3-element structure from one lane of three registers.
ST4 (multiple structures): Store multiple 4-element structures from four registers.
ST4 (single structure): Store single 4-element structure from one lane of four registers.
STNP (SIMD&FP): Store Pair of SIMD&FP registers, with Non-temporal hint.
STP (SIMD&FP): Store Pair of SIMD&FP registers.
STR (immediate, SIMD&FP): Store SIMD&FP register (immediate offset).
STR (register, SIMD&FP): Store SIMD&FP register (register offset).
STUR (SIMD&FP): Store SIMD&FP register (unscaled offset).
SUB (vector): Subtract (vector).
SUBHN, SUBHN2: Subtract returning High Narrow.
SUDOT (by element): Dot product with signed and unsigned integers (vector, by element).
SUQADD: Signed saturating Accumulate of Unsigned value.
TBL: Table vector Lookup.
TBX: Table vector lookup extension.
TRN1: Transpose vectors (primary).
TRN2: Transpose vectors (secondary).
UABA: Unsigned Absolute difference and Accumulate.
UABAL, UABAL2: Unsigned Absolute difference and Accumulate Long.
UABD: Unsigned Absolute Difference (vector).
UABDL, UABDL2: Unsigned Absolute Difference Long.
UADALP: Unsigned Add and Accumulate Long Pairwise.
UADDL, UADDL2: Unsigned Add Long (vector).
UADLP: Unsigned Add Long Pairwise.
UADDLV: Unsigned sum Long across Vector.
UADDW, UADDW2: Unsigned Add Wide.
UCVTF (scalar, fixed-point): Unsigned fixed-point Convert to Floating-point (scalar).
UCVTF (scalar, integer): Unsigned integer Convert to Floating-point (scalar).
UCVTF (vector, fixed-point): Unsigned fixed-point Convert to Floating-point (vector).
UCVTF (vector, integer): Unsigned integer Convert to Floating-point (vector).
UDOT (by element): Dot Product unsigned arithmetic (vector, by element).
UDOT (vector): Dot Product unsigned arithmetic (vector).
UHADD: Unsigned Halving Add.
UHSUB: Unsigned Halving Subtract.
UMAX: Unsigned Maximum (vector).
UMAXP: Unsigned Maximum Pairwise.
UMAXV: Unsigned Maximum across Vector.
UMIN: Unsigned Minimum (vector).
UMINP: Unsigned Minimum Pairwise.
UMINV: Unsigned Minimum across Vector.
UMLSL, UMLSL2 (by element): Unsigned Multiply-Subtract Long (vector, by element).
UMLSL, UMLSL2 (vector): Unsigned Multiply-Subtract Long (vector).
UMMLA (vector): Unsigned 8-bit integer matrix multiply-accumulate (vector).
UMOV: Unsigned Move vector element to general-purpose register.
UMULL, UMULL2 (by element): Unsigned Multiply Long (vector, by element).
UMULL, UMULL2 (vector): Unsigned Multiply long (vector).
UQADD: Unsigned saturating Add.
UQRSHL: Unsigned saturating Rounding Shift Left (register).
UQRSHRN, UQRSHRN2: Unsigned saturating Rounded Shift Right Narrow (immediate).
UQSHL (immediate): Unsigned saturating Shift Left (immediate).
UQSHL (register): Unsigned saturating Shift Left (register).
UQSHRN, UQSHRN2: Unsigned saturating Shift Right Narrow (immediate).
UQSUB: Unsigned saturating Subtract.
UQXTN, UQXTN2: Unsigned saturating extract Narrow.
URECPE: Unsigned Reciprocal Estimate.
URHADD: Unsigned Rounding Halving Add.
URSHL: Unsigned Rounding Shift Left (register).
URSHR: Unsigned Rounding Shift Right (immediate).
URSORTE: Unsigned Reciprocal Square Root Estimate.
URSRRA: Unsigned Rounding Shift Right and Accumulate (immediate).
USDOT (by element): Dot Product with unsigned and signed integers (vector, by element).
USDOT (vector): Dot Product with unsigned and signed integers (vector).
USHL: Unsigned Shift Left (register).
USHLL, USHLL2: Unsigned Shift Left Long (immediate).
USHR: Unsigned Shift Right (immediate).
USMMLA (vector): Unsigned and signed 8-bit integer matrix multiply-accumulate (vector).
USQADD: Unsigned saturating Accumulate of Signed value.
**USRA**: Unsigned Shift Right and Accumulate (immediate).

**USUBL, USUBL2**: Unsigned Subtract Long.

**USUBW, USUBW2**: Unsigned Subtract Wide.

**UXTL, UXTL2**: Unsigned extend Long: an alias of USHLL, USHLL2.

**UZP1**: Unzip vectors (primary).

**UZP2**: Unzip vectors (secondary).

**XAR**: Exclusive OR and Rotate.

**XTN, XTN2**: Extract Narrow.

**ZIP1**: Zip vectors (primary).

**ZIP2**: Zip vectors (secondary).
ABS

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 0 0</th>
<th>0 1 0 1 1 1 0</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
</table>

ABS \(<V>\langle d\rangle, <V>\langle n\rangle\)

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 0 0</th>
<th>0 1 0 1 1 1 0</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
</table>

ABS \(<Vd>.\langle T\rangle, <Vn>.\langle T\rangle\)

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

### Assembler Symbols

<table>
<thead>
<tr>
<th>(&lt;V&gt;)</th>
<th>Is a width specifier, encoded in &quot;size&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;V&gt;)</td>
<td>size</td>
</tr>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(&lt;d&gt;)</th>
<th>Is the number of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;n&gt;)</td>
<td>Is the number of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</td>
</tr>
<tr>
<td>(&lt;Vd&gt;)</td>
<td>Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</td>
</tr>
<tr>
<td>(&lt;T&gt;)</td>
<td>Is an arrangement specifier, encoded in &quot;size:Q&quot;:</td>
</tr>
<tr>
<td>size</td>
<td>Q</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ADD (vector)**

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADD <V><d>, <V><n>, <V><m>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

### Vector

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

### Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<p>| &lt;d&gt;  | Is the number of the SIMD&amp;FP destination register, in the &quot;Rd&quot; field. |
| &lt;n&gt;  | Is the number of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field. |
| &lt;m&gt;  | Is the number of the second SIMD&amp;FP source register, encoded in the &quot;Rm&quot; field. |
| &lt;Vd&gt; | Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field. |
| &lt;T&gt;  | Is an arrangement specifier, encoded in &quot;size:Q&quot;: |</p>
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDHN, ADDHN2

Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see RADDHN.

The ADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the ADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | Q | 0 | 0 | 1 | 1 | 0 | size | 1 | Rm | 0 | 1 | 0 | 0 | 0 | Rn | Rd |
```

ADDHN(2) <Vd>..<Tb>, <Vn>..<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
Q 2
0   [absent]
1   [present]
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size.Q”:

```
size   Q   <Tb>
00     0   8B
00     1   16B
01     0   4H
01     1   8H
10     0   2S
10     1   4S
11     x   RESERVED
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

```
size   <Ta>
00     8H
01     4S
10     2D
11     RESERVED
```

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
   element1 = Elem[operand1, e, 2*esize];
   element2 = Elem[operand2, e, 2*esize];
   if sub_op then
      sum = element1 - element2;
   else
      sum = element1 + element2;
   sum = sum + round_const;
   Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
   • The response of this instruction to asynchronous exceptions does not vary based on:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
ADDP (scalar)

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 0 | size | 1 1 0 0 0 | 1 1 0 1 1 1 0 | Rn | Rd
```

ADDP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize * 2;

Assembler Symbols

<V> Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_ADD, operand, esize);

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ADDP (vector)**

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0  Q  0  0  1  1  1  0  size  1  Rm  1  0  1  1  1  1  Rn  Rd
```

ADDP `<Vd>`..<T>, `<Vn>`..<T>, `<Vm`..<T>

integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;
integer m = `UInt(Rm)`;
if size:Q == '110' then UNDEFINED;
integer esize = 8 << `UInt(size)`;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ADDV

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | O | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rd |

ADDV <V><d>, <Vn>.<T>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;

to integer esize = 8 << UInt(size);
to integer datasize = if Q == '1' then 128 else 64;

Assemble Symbol

- `<V>`: Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<T>`: Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_ADD, operand, esize);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESD

AES single round decryption.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  |

AESD <Vd>.16B, <Vn>.16B

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
result = operand1 EOR operand2;
result = AESInvSubBytes(AESInvShiftRows(result));
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESE

AES single round encryption.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1 | 0  | 0  | 1  | 0  |

D

AESE $<V_d>$.16B, $<V_n>.16B$

integer $d = 	ext{UInt}(R_d)$;
integer $n = 	ext{UInt}(R_n)$;
if !HaveAESEExt() then UNDEFINED;

Assembler Symbols

$<V_d>$ Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.

$<V_n>$ Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = $V[d]$;
bits(128) operand2 = $V[n]$;
bits(128) result;
result = operand1 EOR operand2;
result = AESSubBytes(AESShiftRows(result));

$V[d] = result$;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AESIMC

AES inverse mix columns.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | Rn | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | Rn | 0  | Rd |

AESIMC <Vd>.16B, <Vn>.16B

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

Assembler Symbols

>Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
>Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n];
bits(128) result;
result = AESInvMixColumns(operand);
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESMC

AES mix columns.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  |

D

AESMC <Vd>.16B, <Vn>.16B

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n];
bits(128) result;
result = AESMixColumns(operand);
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.


date 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\hline
0 & Q & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & Rm & 0 & 0 & 0 & 1 & 1 & 1 & 1 & Rn & Rd \\
\hline
\textbf{size}
\hline

\texttt{AND <Vd>., <Vn>., <Vm>.,<T>}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
\end{verbatim}

\textbf{Assembler Symbols}

\begin{description}
\item[<Vd>] Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\item[<T>] Is an arrangement specifier, encoded in "Q":
\begin{center}
\begin{tabular}{ll}
0 & 8B \\
1 & 16B \\
\end{tabular}
\end{center}
\item[Vm>] Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\item[Vn>] Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\end{description}

\textbf{Operation}

\begin{verbatim}
\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(datasize) operand1 = V[n];}
\texttt{bits(datasize) operand2 = V[m];}
\texttt{bits(datasize) result;}
\texttt{result = operand1 AND operand2;}
\texttt{V[d] = result;}
\end{verbatim}

\textbf{Operational information}

If PSTATE.DIT is 1:
\begin{itemize}
\item The execution time of this instruction is independent of:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
**BCAX**

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register. This instruction is implemented only when `FEAT_SHA3` is implemented.

**Advanced SIMD**
*(Armv8.2)*

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0 0 1</td>
</tr>
</tbody>
</table>

**BCAX** `<Vd>`.16B, `<Vn>`.16B, `<Vm>`.16B, `<Va>`.16B

if !`HaveSHA3Ext()` then UNDEFINED;
integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;
integer m = `UInt(Rm)`;
integer a = `UInt(Ra)`;

**Assembler Symbols**

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Va>` is the name of the third SIMD&FP source register, encoded in the "Ra" field.

**Operation**

`AArch64.CheckFPAdvSIMDEnabled()`;

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];

V[d] = Vn EOR (Vm AND NOT(Va));

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Floating-point convert from single-precision to BFloat16 format (scalar) converts the single-precision floating-point value in the 32-bit SIMD&FP source register to BFloat16 format and writes the result in the 16-bit SIMD&FP destination register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception that causes a cumulative exception bit in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR, ID_AA64ISAR1_EL1. BF16 indicates whether this instruction is supported.

Single-precision to BFloat16
(Armv8.6)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | Rn | Rd |

BFCVT <Hd>, <Sn>

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(32) operand = V[n];
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

Elem[result, 0, 16] = FPConvertBF(operand, fpcr);
V[d] = result;
BFCVTN, BFCVTN2

Floating-point convert from single-precision to BFloat16 format (vector) reads each single-precision element in the SIMD&FP source vector, converts each value to BFloat16 format, and writes the results in the lower or upper half of the SIMD&FP destination vector. The result elements are half the width of the source elements.

The BFCVTN instruction writes the half-width results to the lower half of the destination vector and clears the upper half to zero, while the BFCVTN2 instruction writes the results to the upper half of the destination vector without affecting the other bits in the register.

Unlike the BFloat16 multiplication instructions, this instruction honors all of the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR.

Vector single-precision to BFloat16
(Armv8.6)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------|-----------------|
| 0 | Q | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Rn | Rd |

BFCVTN{2} \(<Vd>, <Ta>, <Vn>\), .4S

\[
\text{if} \ \text{!HaveBF16Ext()} \ \text{then} \ \text{UNDEFINED}; \\
\text{integer} \ n = \text{UInt}(Rn); \\
\text{integer} \ d = \text{UInt}(Rd); \\
\text{integer} \ \text{part} = \text{UInt}(Q); \\
\text{integer} \ \text{elements} = 64 \ \text{DIV} \ 16;
\]

Assembler Symbols

2 \(Q\) is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<\(Vd\)> is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<\(Ta\)> is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;(Ta)&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

<\(Vn\)> is the name of the SIMD&FP source register, encoded in the “Rn” field.

Operation

\[
\text{CheckFPAdvSIMDEnabled64}(); \\
\text{bits}(128) \ \text{operand} = \text{\(V[n]\)}; \\
\text{bits}(64) \ \text{result}; \\
\text{for} \ e = 0 \ \text{to} \ \text{elements}-1 \\
\quad \text{Elem}[\text{result, e, 16}] = \text{FPConvertBF}(\text{Elem[operand, e, 32], FPCR[]}); \\
\text{Vpart}[d, \text{part}] = \text{result};
\]

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFDOT (by element)

BFloat16 floating-point dot product (vector, by element). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Each pair of elements in the first source vector is multiplied by the specified pair of elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element of the destination vector that aligns with the pair of BF16 values in the first source vector. The instruction ignores the FPCR and does not update the FPSR exception status.

The BF16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive. ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector
(Armv8.6)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H<index>

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
exteger elements = datasize DIV 32;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
<index> Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the "H:L" fields.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*i+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*i+1, 16];

    bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
    Elem[result, e, 32] = BFAdd(Elem[operand3, e, 32], sum);

V[d] = result;
**BFDOT (vector)**

BFloat16 floating-point dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair; the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element of the destination vector that aligns with the pair of BF16 values in the first source vector. The instruction ignores the FPCR and does not update the FPSR exception status.

### Vector (Armv8.6)

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BFDOT `<Vd>..<Ta>`, `<Vn>..<Tb>`, `<Vm>..<Tb>`

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>` Is an arrangement specifier, encoded in "Q":
  - 0: 2S
  - 1: 4S
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` Is an arrangement specifier, encoded in "Q":
  - 0: 4H
  - 1: 8H
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
for e = 0 to elements-1
  bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
  bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
  bits(16) elt2_a = Elem[operand2, 2*e+0, 16];
  bits(16) elt2_b = Elem[operand2, 2*e+1, 16];
  bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
  Elem[result, e, 32] = BFAdd(Elem[operand3, e, 32], sum);
```

V[d] = result;
BFMLALB, BFMLALT (by element)

BFLOAT16 floating-point widening multiply-add long (by element) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from BFLOAT16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector.

This performs a fused multiply-add without intermediate rounding that honors all of the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR. ID_AA64ISAR1_EL1. BF16 indicates whether this instruction is supported.

Vector
(Armv8.6)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    | Q |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | |

BFMLALB <bt> &lt;Vd&gt;.4S, &lt;Vn&gt;.8H, &lt;Vm&gt;.H{&lt;index&gt;}

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt('0':Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);
integer elements = 128 DIV 32;
integer sel = UInt(Q);

Assembler Symbols

&lt;bt&gt; Is the bottom or top element specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;bt&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
</tr>
</tbody>
</table>

&lt;Vd&gt; Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

&lt;Vn&gt; Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

&lt;Vm&gt; Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

&lt;index&gt; Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) result;
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(32) element2 = Elem[operand2, index, 16]:Zeros(16);
for e = 0 to elements-1
  bits(32) element1 = Elem[operand1, 2*e+sel, 16]:Zeros(16);
  bits(32) addend = Elem[operand3, e, 32];
  Elem[result, e, 32] = BFMulAdd(addend, element1, element2, FPCR[ ]);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFMLALB, BFMLALT (vector)

BFloat16 floating-point widening multiply-add long (vector) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values to the overlapping single-precision elements of the destination vector. This performs a fused multiply-add without intermediate rounding that honors all of the control bits in the FPCR that apply to single-precision arithmetic, including the rounding mode. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR. ID_AA64ISAR1_EL1.BF16 indicates whether these instruction is supported.

Vector
(Armv8.6)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  |

BFMLAL\texttt{<bt>} <Vd>.4S, <Vn>.8H, <Vm>.8H

if !\texttt{HaveBF16Ext}() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer elements = 128 DIV 32;
integer sel = UINT(Q);

Assembler Symbols

\texttt{<bt>} Is the bottom or top element specifier, encoded in "Q":

\begin{array}{c|c}
Q & \texttt{<bt>} \\
\hline
0 & B \\
1 & T \\
\end{array}

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\texttt{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Vm>} Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

\texttt{CheckFPAdvSIMDEnabled64}();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(128) result;

for e = 0 to elements-1

\begin{align*}
\text{bits}(32) & \text{ element1} = \text{Elem}(\text{operand1}, 2*e+\text{sel}, 16):\text{Zeros}(16); \\
\text{bits}(32) & \text{ element2} = \text{Elem}(\text{operand2}, 2*e+\text{sel}, 16):\text{Zeros}(16); \\
\text{bits}(32) & \text{ addend} = \text{Elem}(\text{operand3}, e, 32); \\
\text{Elem}(\text{result}, e, 32) & = \text{BFMulAdd}(\text{addend, element1, element2, FPCR[]});
\end{align*}

V[d] = result;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFMMLA

BFLOAT16 floating-point matrix multiply-accumulate into 2x2 matrix. This instruction multiplies the 2x4 matrix of BF16 values held in the first 128-bit source vector by the 4x2 BF16 matrix in the second 128-bit source vector. The resulting 2x2 single-precision matrix product is then added destructively to the 2x2 single-precision matrix in the 128-bit destination vector. This is equivalent to performing a 4-way dot product per destination element. The instruction ignores the FPCR and does not update the FPSR exception status.

Arm expects that the BFMMLA instruction will deliver a peak BF16 multiply throughput that is at least as high as can be achieved using two BFDOT instructions, with a goal that it should have significantly higher throughput.

Vector
(Armv8.6)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 1 | 1 | 1 | 0 | 1 | 1 | Rn | Rd |

BFMMLA <Vd>.4S, <Vn>.8H, <Vm>.8H

if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) op1 = V[n];
bits(128) op2 = V[m];
bits(128) acc = V[d];
V[d] = BFMatMulAdd(acc, op1, op2);
BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;

Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.
<T> For the 16-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".
<amount> For the 16-bit variant: is the shift amount encoded in "cmode<1>":

<table>
<thead>
<tr>
<th>cmode&lt;1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.
For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Instruction Format]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | Rm | 0  | 0  | 0  | 1  | 1  | Rn | Rd |

size

BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assemble Symbol

$\langle Vd \rangle$ is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$\langle T \rangle$ is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>$\langle T \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

$\langle Vn \rangle$ is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$\langle Vm \rangle$ is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 AND operand2;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];
operand1 = V[d];
operand3 = NOT(V[m]);
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**BIT**

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the *CPACR_EL1, CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| Q | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | Rm | 0 | 0 | 0 | 1 | 1 | 1 | 1 | Rd |
```

**opc2**

BIT <Vd>, <T>, <Vn>, <T>, <Vm>, <T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "Q":
  
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = V[d];
operand3 = V[m];
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BSL

Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Register Bit Representation]

BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = V[m];
operand3 = V[d];
V[d] = operand1 EOR ((operand1 EOR operand4) AND operand3);

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CLS (vector)

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.


count = 0

t = '0'

\textbf{Asm:}

\begin{verbatim}
CLS <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;

<Asm>

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;

debug(X, V);
\end{verbatim}

\textbf{Operational information}

If PSTATE.DIT is 1:
\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
  \item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
    \end{itemize}
\end{itemize}
The values of the NZCV flags.
CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

integer count;
for e = 0 to elements-1
  if countop == CountOp_CLS then
    count = CountLeadingSignBits(Elem[operand, e, esize]);
  else
    count = CountLeadingZeroBits(Elem[operand, e, esize]);
  Elem[result, e, esize] = count<esize-1:0>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 1 1 1 1 0 | size | 1 | Rm | 1 0 0 0 1 1 | Rn | Rd |
```

CMEQ <V><d>, <V><n>, <V><m>

```java
integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
if size != '11' then UNDEFINED;
esize = 8 << UInt(size);
datasize = esize;
elements = 1;
and_test = (U == '0');
```

Vector

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 Q 1 0 1 1 1 0 | size | 1 | Rm | 1 0 0 0 1 1 | Rn | Rd |
```

CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```java
integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
esize = 8 << UInt(size);
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV esize;
and_test = (U == '0');
```

Assembler Symbols

- `<V>` is a width specifier, encoded in “size”:
  ```
<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>
  ```

- `<d>` is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in “size:Q”: 
size  Q  <T>
00  0  8B
00  1  16B
01  0  4H
01  1  8H
10  0  2S
10  1  4S
11  0  RESERVED
11  1  2D

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**CMEQ (zero)**

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd
```

CMEQ `<V><d>, <V><n>, #0`

integer `d = UInt(Rd);`
integer `n = UInt(Rn);`

if `size != '11'` then UNDEFINED;
integer `esize = 8 << UInt(size);`
integer `datasize = esize;`
integer `elements = 1;`

```plaintext
CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector

```plaintext
0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd
```

CMEQ `<Vd>,<T>, <Vn>.<T>, #0`

integer `d = UInt(Rd);`
integer `n = UInt(Rn);`

if `size:Q == '110'` then UNDEFINED;
integer `esize = 8 << UInt(size);`
integer `datasize = if Q == '1' then 128 else 64;`
integer `elements = datasize DIV esize;`

```plaintext
CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Assembler Symbols

`<V>` Is a width specifier, encoded in “size”:
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 8  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | U  | eq |

CMGE `<V><d>`, `<V><n>`, `<V><m>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | Q  | eq |

CMGE `<Xd>.<T>`, `<Xn>.<T>`, `<Xm>.<T>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q != '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

### Assembler Symbols

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Xd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q".
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{n}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Vector

```c
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

Assembler Symbols

```c
<V> Is a width specifier, encoded in “size”:
```
<d>        Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n>        Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd>      Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>        Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>      Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
0 1 0 1 1 1 1 0 | size | Rm
0 0 1 1 0 1 | Rn
```

CMGT <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Vector

```
0 | Q | 0 1 1 1 0 | size | Rm
0 0 1 1 0 | Rn
```

CMGT <Vd>..<T>, <Vn>..<T>, <Vm>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”: 

CMGT (register)  Page 633
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|---------------------|
| 0 1 0 1 1 1 0       | size 1 0 0 0 0      | 0 1 0 0 0 1 1 0     |
|                      | Rn                | Rd                  |

**CMGT <V><d>, <V><n>, #0**

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

**Vector**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|---------------------|
| 0 Q 0 1 1 1 1 0     | size 1 0 0 0 0      | 0 1 0 0 0 1 1 0     |
|                      | Rn                | Rd                  |

**CMGT <Vd>.,<T>, <Vn>.,<T>, #0**

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**CompareOp** comparison;
case op:U of
when '00' comparison = CompareOp_GT;
when '01' comparison = CompareOp_GE;
when '10' comparison = CompareOp_EQ;
when '11' comparison = CompareOp_LE;

**Assembler Symbols**

**<V>** Is a width specifier, encoded in “size”: 
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMHI (register)

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```
0 1 1 1 1 1 1 0 | size 1 | Rd
0 0 1 1 0 1 | Rn
1
```

CMHI <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

**Vector**

```
0 Q 1 0 1 1 1 0 | size 1 | Rd
0 0 1 1 0 1 | Rn
1
```

CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');

**Assembler Symbols**

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
b bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMHS (register)

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 1 | 1 1 1 1 1 0 | size | 1 | Rm | 0 1 1 1 1 1 | Rn | Rd
   U |    |     |    | eq

CMHS <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
if size != '11' then UNDEFINED;
esize = 8 << UInt(size);
datasize = esize;
elements = 1;
unsigned = (U == '1');
cmp_eq = (eq == '1');

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 1 1 1 1 1 | Rn | Rd
   U |    |    |    | eq

CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
esize = 8 << UInt(size);
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV esize;
unsigned = (U == '1');
cmp_eq = (eq == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”: 
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMLE (zero)

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------|-----------|-----------|-----------|-----------|-----------|
| 0 1 1 1 1 1 1 0   | size      | 1 0 0 0 0 | 0 1 0 1 1 0 | Rd         |                   |
| U                  | op        | Rn        |           |            |                   |

CMLE <V><d>, <V><n>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-------------------|-----------|-----------|-----------|-----------|-----------|
| 0 Q 1 0 1 1 1 0   | size      | 1 0 0 0 0 | 0 1 0 1 1 0 | Rd         |                   |
| U                  | op        | Rn        |           |            |                   |

CMLE <Vd>..<T>, <Vn>..<T>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler Symbols

<V> Is a width specifier, encoded in “size”: 
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
</tr>
<tr>
<td>00</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>01</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
   element = SInt(Elem[operand, e, esize]);
   case comparison of
      when CompareOp_GT test_passed = element > 0;
      when CompareOp_GE test_passed = element >= 0;
      when CompareOp_EQ test_passed = element == 0;
      when CompareOp_LE test_passed = element <= 0;
      when CompareOp_LT test_passed = element < 0;
   Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the CPACR_EL1, CPTR EL2, and CPTR EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>0 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

CMLT <V><d>, <V><n>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>0 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

CMLT <Vd>.<T>, <Vn>.<T>, #0

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

**Assembler Symbols**

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**CMTST**

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 1  | 1  | 1  | 0  | size| 1  | 1  | Rm | 1  | 0  | 0  | 0  | 1  | 1  |
| Rn | Rd |

CMTST `<V><d>, <V><n>, <V><m>`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 * UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');

### Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0  | 0  | 1  | 1  | 1  | 0  | size| 1  | 1  | Rm | 1  | 0  | 0  | 0  | 1  | 1  |
| Rn | Rd |

CMTST `<Vd><T>, <Vn><T>, <Vm><T>`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 * UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');

### Assembler Symbols

- **<V>** is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<d>** is the number of the SIMD&FP destination register, in the "Rd" field.
- **<n>** is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- **<m>** is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Vd>** is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** is an arrangement specifier, encoded in “size:Q”: 
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | Rd | Rn |

CNT <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '00' then UNDEFINED;
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

bits(datasize) result;

integer count;

for e = 0 to elements-1
  count = BitCount(Elem[operand, e, esize]);
  Elem[result, e, esize] = count<esize-1:0>;
  V[d] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
DUP (element)

Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar).

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 1 0 1 1 1 1 0 0 0 0 | imm5 | 0 0 0 0 0 1 | Rn | Rd |

DUP <V><d>, <Vn>.<T>[<index>]

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 Q 0 1 1 1 1 0 0 0 0 | imm5 | 0 0 0 0 0 1 | Rn | Rd |

DUP <Vd>.<T>, <Vn>.<Ts>[<index>]

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

For the scalar variant: is the element width specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

For the vector variant: is an arrangement specifier, encoded in “imm5:Q”:
<Ts> Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>Q</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>xxxx1</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>xxxx10</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<V> Is the destination width specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
DUP <Vd>.<T>, <R><n>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;
// imm5<4:size+1> is IGNORED
if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
in integer datasetsize = if Q == '1' then 128 else 64;
in integer elements = datasetsize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "imm5:Q":

<table>
<thead>
<tr>
<th>imm5</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>xxxx1</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>xxx10</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>xxx10</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<R> Is the width specifier for the general-purpose source register, encoded in "imm5":

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(datasize) result;
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
EOR3

Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.
This instruction is implemented only when `FEAT_SHA3` is implemented.

**Advanced SIMD**
(*Armv8.2*)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | Rm | 0  | Ra | Rn | Rd |

EOR3 `<Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B`

if `!HaveSHA3Ext()` then UNDEFINED;
integer d = `UInt(Rd)`;
integer n = `UInt(Rn)`;
integer m = `UInt(Rm)`;
integer a = `UInt(Ra)`;

**Assembler Symbols**

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
`<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
`<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
`<Va>` Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

**Operation**

`AArch64.CheckFPAdvSIMDEnabled()`;

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Va = V[a];
V[d] = Vn EOR Vm EOR Va;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
EOR (vector)

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register. Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
EOR <Vd>..<T>, <Vn>..<T>, <Vm>..<T>
\end{verbatim}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
\end{verbatim}

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "Q":
  \begin{tabular}{c|c}
  \hline
  Q & <T> \\
  \hline
  0 & 8B \\
  1 & 16B \\
  \hline
  \end{tabular}
- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = \texttt{V}[n];
operand1 = \texttt{V}[m];
operand2 = \texttt{Zeros}();
operand3 = \texttt{Ones}();
\texttt{V}[d] = operand1 \texttt{EOR ((operand2 EOR operand4) AND operand3);};
\end{verbatim}

**Operational information**

If PSTATE.DIT is 1:
\begin{itemize}
  \item The execution time of this instruction is independent of:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
  \item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
      \item The values of the data supplied in any of its registers.
      \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}
Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for $Q = 0$ and $imm4<2:0> = 3$.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
EXT <Vd>, <T>, <Vn>, <T>, <Vm>, #<index>
```

```markdown
<table>
<thead>
<tr>
<th>(Q)</th>
<th>(imm4&lt;3&gt;)</th>
<th>(&lt;index&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>(imm4&lt;2:0&gt;)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>(RESERVED)</td>
</tr>
<tr>
<td>1</td>
<td>(x)</td>
<td>(imm4)</td>
</tr>
</tbody>
</table>
```

**Asmblemmer Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "Q":
  - 0: \(8B\)
  - 1: \(16B\)
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<index>` Is the lowest numbered byte element to be extracted, encoded in "Q:imm4":

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m];
bits(datasize) lo = V[n];
bits(datasize*2) concat = hi:lo;
V[d] = concat<position+ datasize-1:position>;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

**Scalar half precision**
(Armv8.2)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 1 1 0 | Rm | 0 0 0 1 0 1 | Rn | Rd
```

```
FABD <Hd>, <Hn>, <Hm>
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

**Scalar single-precision and double-precision**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 | sz | 1 | Rm | 1 1 0 1 0 1 | Rn | Rd
```

```
FABD <V>d>, <V>n>, <V>m>
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

**Vector half precision**
(Armv8.2)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 0 1 1 0 | Rm | 0 0 0 1 0 1 | Rn | Rd
```

```
U
```
FABD <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

Vector single-precision and double-precision

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;
V[d] = result;
FABS (vector)

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

---

**Half-precision**

(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FABS <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

---

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FABS <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

---

**Assembler Symbols**

| <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <T> For the half-precision variant: is an arrangement specifier, encoded in "Q": |
| Q <T> |
| 0 4H |
| 1 8H |

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

| sz Q <T> |
| 0 0 2S |
| 0 1 4S |
| 1 0 RESERVED |
| 1 1 2D |

| <Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field. |
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d] = result;
FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Hexadecimal representation of FABS instruction](image)

**Half-precision (ftype == 11) (Armv8.2)**

\[ \text{FABS } <Hd>, <Hn> \]

**Single-precision (ftype == 00)**

\[ \text{FABS } <Sd>, <Sn> \]

**Double-precision (ftype == 01)**

\[ \text{FABS } <Dd>, <Dn> \]

integer \( d = \) UInt(Rd);
integer \( n = \) UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

- \(<Dd>\) Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<Dn>\) Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- \(<Hd>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<Hn>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- \(<Sd>\) Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<Sn>\) Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\[ \text{CheckFPAdvSIMDEnabled64}(); \]
\[ \text{FPCRType } fpcr = \text{FPCR}[]; \]
\[ \text{boolean merge = } \text{IsMerging}(fpcr); \]
\[ \text{bits(128) result } = \text{if merge then } V[d] \text{ else Zeros}(); \]

\[ \text{bits(esize) operand } = V[n]; \]
\[ \text{Elem}[\text{result, 0, esize}] = \text{FPAbs(operand)}; \]
\[ V[d] = \text{result}; \]
**FACGE**

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1, CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

### Scalar half precision (Armv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FACGE <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

```c
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>E</td>
<td>ac</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FACGE <Hd>, <Hn>, <Hm>
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;
case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector half precision
(Armv8.2)

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;
case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector single-precision and double-precision
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(eseize) element1;
bits(eseize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
FACGT

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 1 | 1 | Rn | Rd |

U E ac

FACGT <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Scalar single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | sz | 1 | Rm | 1 | 1 | 1 | 0 | 1 | 1 | Rn | Rd |

U E ac
FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Vector half precision
(Armv8.2)

FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Vector single-precision and double-precision

if !HaveFP16Ext() then UNDEFINED;
integer \( d = \text{UInt}(R_d) \);
integer \( n = \text{UInt}(R_n) \);
integer \( m = \text{UInt}(R_m) \);
if \( \text{sz:Q} == '10' \) then UNDEFINED;
integer \( \text{esize} = 32 \ll \text{UInt}(\text{sz}) \);
integer \( \text{datasize} = \text{if } Q == '1' \text{ then } 128 \text{ else } 64 \);
integer \( \text{elements} = \text{datasize} \div \text{esize} \);
\text{CompareOp} \ cmp;
\text{boolean} \ \text{abs};

\text{case } E:U:a: \text{ of}
\quad \text{when } '000' \ cmp = \text{CompareOp_EQ}; \ \text{abs} = \text{FALSE};
\quad \text{when } '010' \ cmp = \text{CompareOp_GE}; \ \text{abs} = \text{FALSE};
\quad \text{when } '011' \ cmp = \text{CompareOp_GT}; \ \text{abs} = \text{TRUE};
\quad \text{when } '110' \ cmp = \text{CompareOp_GT}; \ \text{abs} = \text{FALSE};
\quad \text{when } '111' \ cmp = \text{CompareOp_GT}; \ \text{abs} = \text{TRUE};
\quad \text{otherwise } \text{UNDEFINED};

\textbf{Assembler Symbols}

\texttt{<Hd>} \quad \text{Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
\texttt{<Hn>} \quad \text{Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.}
\texttt{<Hm>} \quad \text{Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.}

\texttt{<V>} \quad \text{Is a width specifier; encoded in "sz":}
\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{sz} & \texttt{<V>} \\
\hline
0 & S \\
1 & D \\
\hline
\end{tabular}
\end{center}

\texttt{<d>} \quad \text{Is the number of the SIMD&FP destination register, in the "Rd" field.}
\texttt{<n>} \quad \text{Is the number of the first SIMD&FP source register, encoded in the "Rn" field.}
\texttt{<m>} \quad \text{Is the number of the second SIMD&FP source register, encoded in the "Rm" field.}

\texttt{<Vd>} \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}
\texttt{<T>} \quad \text{For the half-precision variant: is an arrangement specifier, encoded in "Q":}
\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{Q} & \texttt{<T>} \\
\hline
0 & 4H \\
1 & 8H \\
\hline
\end{tabular}
\end{center}

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\texttt{sz} & \texttt{Q} & \texttt{<T>} \\
\hline
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & \text{RESERVED} \\
1 & 1 & 2D \\
\hline
\end{tabular}
\end{center}

\texttt{<Vn>} \quad \text{Is the name of the first SIMD&FP source register, encoded in the "Rn" field.}
\texttt{<Vm>} \quad \text{Is the name of the second SIMD&FP source register, encoded in the "Rm" field.}
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(eseize) element1;
bis(eseize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bis(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
   element1 = Elem[operand1, e, esize];
   element2 = Elem[operand2, e, esize];
   if abs then
      element1 = FPAbs(element1);
      element2 = FPAbs(element2);
   case cmp of
      when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
      when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
      when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
   end case
   Elem[result, e, esize] = if test_passed then Ones() else Zeros();

V[d] = result;
```
FADD (vector)

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 1 | Rn | Rd |
| U |

FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|
| 0 | 0 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |
| U |

FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == ‘10’ then UNDEFINED;
integer esize = 32 <<UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean pair = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Half-precision (ftype == 11)  
(Armv8.2)

FADD <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FADD <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FADD <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypetype fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPAdd(operand1, operand2, fpcr);
V[d] = result;
FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**
(Armv8.2)

```
0 1 0 1 1 1 1 0 0 0 1 1 0 1 1 1 0 0 1 1 0 1 1 0
sz
Rn  Rd
```

FADDP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
n = UInt(Rn);
esize = 16;
datasize = 32;

**Single-precision and double-precision**

```
0 1 1 1 1 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0 1 1 0
sz
Rn  Rd
```

FADDP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
n = UInt(Rn);
esize = 32;
datasize = 64;

**Assembler Symbols**

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

```
sz   <V>
0   H
1   RESERVED
```

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

```
sz   <V>
0   S
1   D
```

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz”:

```
sz   <T>
0   2H
1   RESERVED
```

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

FADDP (scalar)
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FADD, operand, esize);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FADDP (vector)

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 1 | Rn | Rd |
| U |

FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

**Single-precision and double-precision**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 1 | 0 | 1 | Rn | Rd |
| U |

FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');

Assembler Symbols

**<Vd>**

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>**

For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCADD**

Floating-point Complex Add.
This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Vector**
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | size | 0  | 0  | 1  | 1  | 1  | 0  | rot | 0  | 0  | 1  | Rm | 1  | 1  | 1  | Rn | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |

FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>1</td>
<td>270</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '0'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element3 = Elem[operand2, e*2, esize];
        when '1'
            element1 = Elem[operand2, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
    Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR[]);
    Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCCMP**

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the \( \text{PSTATE} \).\{N, Z, C, V\} flags. If the condition does not pass then the \( \text{PSTATE} \).\{N, Z, C, V\} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in \( \text{FPCR} \), the exception results in either a flag being set in \( \text{FPSR} \), or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the \( \text{CPACR\_EL1} \), \( \text{CPTR\_EL2} \), and \( \text{CPTR\_EL3} \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```plaintext
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ftype</td>
<td>1</td>
<td>Rm</td>
<td>cond</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>0</td>
<td>nzcv</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Half-precision (ftype == 11)**
(Armv8.2)

\[
\text{FCCMP } \langle \text{Hn} \rangle, \langle \text{Hm} \rangle, \#\langle \text{nzcv} \rangle, \langle \text{cond} \rangle
\]

**Single-precision (ftype == 00)**

\[
\text{FCCMP } \langle \text{Sn} \rangle, \langle \text{Sm} \rangle, \#\langle \text{nzcv} \rangle, \langle \text{cond} \rangle
\]

**Double-precision (ftype == 01)**

\[
\text{FCCMP } \langle \text{Dn} \rangle, \langle \text{Dm} \rangle, \#\langle \text{nzcv} \rangle, \langle \text{cond} \rangle
\]

integer \( n = \text{UInt}(\text{Rn}) \);

integer \( m = \text{UInt}(\text{Rm}) \);

integer datasize;

```plaintext
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if \( \text{HaveFP16Ext}() \) then
      datasize = 16;
    else
      UNDEFINED;

bits(4) flags = nzcv;
```

**Assembler Symbols**

\(<\text{Dn}>\) Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Dm}>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

\(<\text{Hn}>\) Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Hm}>\) Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

\(<\text{Sn}>\) Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Sm}>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

\(<\text{nzcv}>\) Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.

\(<\text{cond}>\) Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

```c
CheckFPAdvSIMEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];
if ConditionHolds(cond) then
    flags = FPCmp(operand1, operand2, FALSE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.
FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*.\{N, Z, C, V\} flags. If the condition does not pass then the *PSTATE*.\{N, Z, C, V\} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1    | Rm   | cond | 0  | 1  | Rn | 1   | nzcv |

**Half-precision (ftype == 11)
(ARMv8.2)**

FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>

**Single-precision (ftype == 00)**

FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>

**Double-precision (ftype == 01)**

FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>

```plaintext
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;

case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

    bits(4) flags = nzcv;
```

**Assembler Symbols**

- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<nzcv>` Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
- `<cond>` Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];
if ConditionHolds(cond) then
    flags = FPCmpared(operand1, operand2, TRUE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.
Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision

(Armv8.2)

```
0 1 0 1 1 1 1 0 0 1 0  Rm 0 0 1 0 0 1  Rn  Rd
```

FCMEQ <Hd>, <Hn>, <Hm>

```plaintext
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;
```

```
case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;
```

### Scalar single-precision and double-precision

```
0 1 0 1 1 1 1 0 0 | sz | Rm 1 1 1 0 0 1  Rn  Rd
```

```plaintext
FCMEQ <Hd>, <Hn>, <Hm>

```
integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
integer esize = 32 \ll \text{UInt}(sz);
integer datasize = esize;
integer elements = 1;
\textbf{CompareOp} cmp;
boolean abs;

case E:U:ac of
  when '000' \(\text{cmp} = \text{CompareOp\_EQ}\); abs = FALSE;
  when '010' \(\text{cmp} = \text{CompareOp\_GE}\); abs = FALSE;
  when '011' \(\text{cmp} = \text{CompareOp\_GE}\); abs = TRUE;
  when '110' \(\text{cmp} = \text{CompareOp\_GT}\); abs = FALSE;
  when '111' \(\text{cmp} = \text{CompareOp\_GT}\); abs = TRUE;
  otherwise UNDEFINED;

\textbf{Vector half precision}
\textbf{(Armv8.2)}

\begin{array}{cccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & \text{Rm} & 0 & 0 & 1 & 0 & 0 & 1 & \text{Rn} & \text{Rd} & \text{ac} & \text{E} & \text{U} & \text{ac} & \text{E} & \text{U} & \text{ac}& \end{array}

\textbf{FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

if !\text{HaveFP16Ext}() then UNDEFINED;

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize \div esize;
\textbf{CompareOp} cmp;
boolean abs;

case E:U:ac of
  when '000' \(\text{cmp} = \text{CompareOp\_EQ}\); abs = FALSE;
  when '010' \(\text{cmp} = \text{CompareOp\_GE}\); abs = FALSE;
  when '011' \(\text{cmp} = \text{CompareOp\_GE}\); abs = TRUE;
  when '110' \(\text{cmp} = \text{CompareOp\_GT}\); abs = FALSE;
  when '111' \(\text{cmp} = \text{CompareOp\_GT}\); abs = TRUE;
  otherwise UNDEFINED;

\textbf{Vector single-precision and double-precision}

\begin{array}{cccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & \text{Rm} & 1 & 1 & 1 & 0 & 0 & 1 & \text{Rn} & \text{Rd} & \text{ac} & \text{E} & \text{U} & \text{ac} & \text{E} & \text{U} & \text{ac}& \end{array}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;
case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\texttt{CheckFPAvSIMDEnabled64()};
bits(\text{datasize}) \text{operand1} = V[n];
bits(\text{datasize}) \text{operand2} = V[m];

bits(\text{esize}) \text{element1};
bits(\text{esize}) \text{element2};
boolean \text{test\_passed};
\text{FPCRType} \text{fpcr} = \text{FPCR[]};
boolean \text{merge} = \text{elements} == 1 \&\& \text{IsMerging}(\text{fpcr});
bits(128) \text{result} = \text{if} \text{merge} \text{then} V[m] \text{else} Zeros();

\text{for} \ e = 0 \ \text{to} \ \text{elements}-1
  \ \text{element1} = \text{Elem}[\text{operand1}, \ e, \ \text{esize}];
  \ \text{element2} = \text{Elem}[\text{operand2}, \ e, \ \text{esize}];
  \ \text{if} \ \text{abs} \ \text{then}
    \ \text{element1} = \text{FPAbs}(\text{element1});
    \ \text{element2} = \text{FPAbs}(\text{element2});
  \ \text{case cmp of}
    \text{when} \ \text{CompareOp\_EQ} \ \text{test\_passed} = \text{FPCompareEQ}(\text{element1}, \ \text{element2}, \ \text{fpcr});
    \text{when} \ \text{CompareOp\_GE} \ \text{test\_passed} = \text{FPCompareGE}(\text{element1}, \ \text{element2}, \ \text{fpcr});
    \text{when} \ \text{CompareOp\_GT} \ \text{test\_passed} = \text{FPCompareGT}(\text{element1}, \ \text{element2}, \ \text{fpcr});
    \text{Elem}[\text{result}, \ e, \ \text{esize}] = \text{if} \ \text{test\_passed} \ \text{then} \ \text{Ones()} \ \text{else} \ \text{Zeros()};
\text{V[d]} = \text{result};
FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(Armv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  0 |  1 |  1 |  1 |  1 |  1 |  0 |  1 |  1 |  1 |  1 |  1 |  0 |  0 |  0 |  1 |  1 |  0 |  1 |  1 |  0 |       |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|  op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

FCMEQ <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
    case op:U of
        when '00' comparison = CompareOp_GT;
        when '01' comparison = CompareOp_GE;
        when '10' comparison = CompareOp_EQ;
        when '11' comparison = CompareOp_LE;
```

Scalar single-precision and double-precision

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  0 |  1 |  1 |  1 |  1 |  1 |  0 |  1 |  1 |  0 |  0 |  0 |  0 |  0 |  0 |  1 |  1 |  0 |  1 |  1 |  0 |       |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|  op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

FCMEQ <V<d>, <V<n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
    case op:U of
        when '00' comparison = CompareOp_GT;
        when '01' comparison = CompareOp_GE;
        when '10' comparison = CompareOp_EQ;
        when '11' comparison = CompareOp_LE;
Vector half precision
(Armv8.2)

31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd
U   op

FCMEQ <Vd>..<T>, <Vn>..<T>, #0.0
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | Rn | Rd
U   op

FCMEQ <Vd>..<T>, <Vn>..<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”: #0.0
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: \textit{Scalar half precision}, \textit{Scalar single-precision and double-precision}, \textit{Vector half precision} and \textit{Vector single-precision and double-precision}

**Scalar half precision**

(\textit{Armv8.2})

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 | 0 1 0 | Rm      | 0 0 1 0 0 1 | Rn      | Rd
\end{verbatim}

FCMGE <Hd>, <Hn>, <Hm>

if !\textit{HaveFP16Ext}() then UNDEFINED;

integer d = \textit{UInt}(Rd);
integer n = \textit{UInt}(Rn);
integer m = \textit{UInt}(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
\textit{CompareOp} cmp;
boolean abs;

\textbf{case E:U:ac of}

\begin{verbatim}
when '000' cmp = \textit{CompareOp\_EQ}; abs = FALSE;
when '010' cmp = \textit{CompareOp\_GE}; abs = FALSE;
when '011' cmp = \textit{CompareOp\_GE}; abs = TRUE;
when '110' cmp = \textit{CompareOp\_GT}; abs = FALSE;
when '111' cmp = \textit{CompareOp\_GT}; abs = TRUE;
otherwise UNDEFINED;
\end{verbatim}

**Scalar single-precision and double-precision**

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 | 0 | sz 1 | Rm      | 1 1 1 0 0 1 | Rn      | Rd
\end{verbatim}

FCMGE (register)
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector half precision
(Armv8.2)

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector single-precision and double-precision
FCMGE \textlt{Vd}.<T>, \textlt{Vn}.<T>, \textlt{Vm}.<T>

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

\textbf{CompareOp} cmp;
boolean abs;

\textbf{case E:U:ac of}
  when '000' cmp = \textbf{CompareOp_EQ}; abs = FALSE;
  when '010' cmp = \textbf{CompareOp_GE}; abs = FALSE;
  when '011' cmp = \textbf{CompareOp_GE}; abs = TRUE;
  when '110' cmp = \textbf{CompareOp_GT}; abs = FALSE;
  when '111' cmp = \textbf{CompareOp_GT}; abs = TRUE;
  otherwise UNDEFINED;
\end{verbatim}

\textbf{Assembler Symbols}

\textbf{<Hd>} Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\textbf{<Hn>} Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
\textbf{<Hm>} Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
\textbf{<V>} Is a width specifier, encoded in "sz":
\begin{center}
\begin{tabular}{c|c}
\textbf{sz} & \textbf{V} \\
0 & S \\
1 & D \\
\end{tabular}
\end{center}

\textbf{<d>} Is the number of the SIMD&FP destination register, in the "Rd" field.
\textbf{<n>} Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
\textbf{<m>} Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
\textbf{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\textbf{<T>} For the half-precision variant: is an arrangement specifier, encoded in "Q":
\begin{center}
\begin{tabular}{c|c}
\textbf{Q} & \textbf{T} \\
0 & 4H \\
1 & 8H \\
\end{tabular}
\end{center}

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":
\begin{center}
\begin{tabular}{c|c|c}
\textbf{sz} & \textbf{Q} & \textbf{T} \\
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & RESERVED \\
1 & 1 & 2D \\
\end{tabular}
\end{center}

\textbf{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\textbf{<Vm>} Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = V[n];
bias(datasize) operand2 = V[m];

bits(ese) element1;
bias(ese) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bias(128) result = if merge then V[m] else Zeros();

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  if abs then
    element1 = FPAbs(element1);
    element2 = FPAbs(element2);
  case cmp of
    when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
    when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
    when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
  Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision.

### Scalar half precision

(ARMv8.2)

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

```
U
```

```
op
```

```
FCMGE <Hd>, <Hn>, #0.0
```

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
```
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

### Scalar single-precision and double-precision

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

```
U
```

```
op
```

```
FCMGE <V><d>, <V><n>, #0.0
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
```
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```
Vector half precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|
| \text{0}\,\text{Q}\,1\,0\,1\,1\,1\,0 \,1 \,1 \,1 \,1 \,0 \,0 \,0 \,1 \,1 \,0 \,0 \,1 \,0 | \text{Rn} | \text{Rd} |

U\quad op

FCMGE \quad <Vd>.<T>, <Vn>.<T>, \#0.0

if \text{!HaveFP16Ext()} \text{ then UNDEFINED;}

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

integer esize = 16;
integer datasize = if Q == '1' \text{ then 128 \text{ else 64;}}
integer elements = datasize \text{ DIV esize;}

\texttt{CompareOp} \quad \text{comparison;}

\text{case op}:U \text{ of}

\quad \text{when '00' \text{ comparison} = \text{CompareOp_GT};}
\quad \text{when '01' \text{ comparison} = \text{CompareOp_GE};}
\quad \text{when '10' \text{ comparison} = \text{CompareOp_EQ};}
\quad \text{when '11' \text{ comparison} = \text{CompareOp_LE};}

Vector single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|
| \text{0}\,\text{Q}\,1\,0\,1\,1\,1\,0 \,1 \,| \text{sz}\,1 \,0 \,0 \,0 \,0 \,0 \,1 \,1 \,0 \,0 \,1 \,0 | \text{Rn} | \text{Rd} |

U\quad op

FCMGE \quad <Vd>.<T>, <Vn>.<T>, \#0.0

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

if sz:Q == '10' \text{ then UNDEFINED;}
integer esize = 32 \text{ \\ll \text{ UInt}(sz);}
integer datasize = if Q == '1' \text{ then 128 \text{ else 64;}}
integer elements = datasize \text{ DIV esize;}

\texttt{CompareOp} \quad \text{comparison;}

\text{case op}:U \text{ of}

\quad \text{when '00' \text{ comparison} = \text{CompareOp_GT};}
\quad \text{when '01' \text{ comparison} = \text{CompareOp_GE};}
\quad \text{when '10' \text{ comparison} = \text{CompareOp_EQ};}
\quad \text{when '11' \text{ comparison} = \text{CompareOp_LE};}

Assembler Symbols

\texttt{<Hd>} \quad \text{Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
\texttt{<Hn>} \quad \text{Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
\texttt{<V>} \quad \text{Is a width specifier, encoded in “sz”:}

\begin{tabular}{c|c}
\texttt{sz} & \texttt{<V>} \\
\hline
0 & S \\
1 & D \\
\end{tabular}

\texttt{<d>} \quad \text{Is the number of the SIMD&FP destination register, encoded in the "Rd" field.}
\texttt{<n>} \quad \text{Is the number of the SIMD&FP source register, encoded in the "Rn" field.}
\texttt{<Vd>} \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}
\texttt{<T>} \quad \text{For the half-precision variant: is an arrangement specifier, encoded in “Q”:}
For the single-precision and double-precision variant: Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{n}> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPA dvSIMDv Enabled 64();
bins(datasize) operand = \text{V}[n];
bins(datasize) result;
bins(esize) zero = FPZero('0');
bins(esize) element;
boolean test_passed;
for e = 0 to elements-1
   element = \text{Elem}[operand, e, esize];
   case comparison of
      when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);  
      when CompareOp GE test_passed = FPCompareGE(element, zero, FPCR[]);   
      when CompareOp EQ test_passed = FPCompareEQ(element, zero, FPCR[]);  
      when CompareOp LE test_passed = FPCompareGE(zero, element, FPCR[]);  
      when CompareOp LT test_passed = FPCompareGT(zero, element, FPCR[]);
         \text{Elem}[result, e, esize] = if test_passed then \text{Ones}() else \text{Zeros}();
    \text{V}[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMGT (register)**

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

**Scalar half precision**

*(Armv8.2)*

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  | E  | ac |

FCMGT <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

**Scalar single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 0  | 1  | Rn | Rd |
| U  | E  | ac |
FCMGT \langle V \rangle \langle d \rangle, \langle V \rangle \langle n \rangle, \langle V \rangle \langle m \rangle

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 \ll UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector half precision

(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[\begin{array}{cccccc|cccccc|cccccc}
0 & Q & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & Rn & 0 & 0 & 1 & 0 & 0 & 1 & Rd
U & E & ac
\end{array}\]

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
  otherwise UNDEFINED;

Vector single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[\begin{array}{cccccc|cccccc|cccccc}
0 & Q & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & Rn & 1 & 1 & 1 & 0 & 0 & 1 & Rd
U & E & ac
\end{array}\]

FCMGT (register)
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = CompareOp_EQ; abs = FALSE;
  when '010' cmp = CompareOp_GE; abs = FALSE;
  when '011' cmp = CompareOp_GE; abs = TRUE;
  when '110' cmp = CompareOp_GT; abs = FALSE;
  when '111' cmp = CompareOp_GT; abs = TRUE;
otherwise UNDEFINED;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Q> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMEnabled64();

bits(datasize) operand1 = v[n];
bits(datasize) operand2 = v[m];

bits(eseze) element1;
bits(eseze) element2;
boolean test_passed;  
FPCRType fpocr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then v[m] else Zeros();

for e = 0 to elements-1
    element1 = Elem[operand1, e, e;size];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FP Abs(element1);
        element2 = FP Abs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FP CompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FP CompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FP CompareGT(element1, element2, fpcr);
        Elem[result, e, e;size] = if test_passed then Ones() else Zeros();

v[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMGT (zero)**

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision
**((Armv8.2))**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 1 1 1 1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**U**

<table>
<thead>
<tr>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
</tr>
</tbody>
</table>

**FCMGT <Hd>, <Hn>, #0.0**

if !**HaveFP16Ext**() then UNDEFINED;

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison;

case op:U of
  when '00' comparison = **CompareOp_GT**;
  when '01' comparison = **CompareOp_GE**;
  when '10' comparison = **CompareOp_EQ**;
  when '11' comparison = **CompareOp_LE**;

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 0 1 sz 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**U**

<table>
<thead>
<tr>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</td>
</tr>
</tbody>
</table>

**FCMGT <V><d>, <V><n>, #0.0**

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);

integer esize = 32 << **UInt**(sz);
integer datasize = esize;
integer elements = 1;

**CompareOp** comparison;

case op:U of
  when '00' comparison = **CompareOp_GT**;
  when '01' comparison = **CompareOp_GE**;
  when '10' comparison = **CompareOp_EQ**;
  when '11' comparison = **CompareOp_LE**;
### Vector half precision

**Armv8.2**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |

```plaintext
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
```

### Assembler Symbols

- `<Hd>`: Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>`: Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<V>`: Is a width specifier, encoded in "sz":
  - sz <V>
    - 0: S
    - 1: D
- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<n>`: Is the number of the SIMD&FP source register, encoded in the "Rn" field.
- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the half-precision variant: is an arrangement specifier, encoded in "Q":
  - FCMGT (zero)
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
        Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCMLA (by element)**

Floating-point Complex Multiply Accumulate (by element).
This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with
the more significant element holding the imaginary part of the number and the less significant element holding the
real part of the number. Each element holds a floating-point value. It performs the following computation on complex
numbers from the first source register and the destination register with the specified complex number from the
second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is
  rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a
    rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation
    was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination
  register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.
This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in
either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point
exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

**Vector**
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | L  | M  | Rm | 0  | rot | 1  | H  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

(size == 01)

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

(size == 10)

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”: 
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in "size:H:L":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<rotate> Is the rotation, encoded in “rot”:  

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;rotate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>180</td>
</tr>
<tr>
<td>11</td>
<td>270</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand2, index*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, index*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, index*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, index*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, index*2+1, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, index*2, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, index*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, index*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];
    
    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMLA

Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(Armv8.3)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | size| 0  | Rm | 1  | 1  | 0  | rot| 1  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

FCMLA \(<Vd>\).<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

if \(!\text{HaveFCADDExt}()\) then \text{UNDEFINED};
integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
if size == '00' then \text{UNDEFINED};
if Q == '0' && size == '11' then \text{UNDEFINED};
integer esize = 8 << \text{UInt}(size);
if \(!\text{HaveFP16Ext}()\) && esize == 16 then \text{UNDEFINED};
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
\(<\text{rotate}>\) Is the rotation, encoded in “rot”: 
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
  case rot of
    when '00'
      element1 = Elem[operand2, e*2, esize];
      element2 = Elem[operand1, e*2, esize];
      element3 = Elem[operand2, e*2+1, esize];
      element4 = Elem[operand1, e*2, esize];
    when '01'
      element1 = FPNeg(Elem[operand2, e*2+1, esize]);
      element2 = Elem[operand1, e*2+1, esize];
      element3 = Elem[operand2, e*2, esize];
      element4 = Elem[operand1, e*2+1, esize];
    when '10'
      element1 = FPNeg(Elem[operand2, e*2, esize]);
      element2 = Elem[operand1, e*2, esize];
      element3 = FPNeg(Elem[operand2, e*2+1, esize]);
      element4 = Elem[operand1, e*2+1, esize];
    when '11'
      element1 = Elem[operand2, e*2+1, esize];
      element2 = Elem[operand1, e*2+1, esize];
      element3 = FPNeg(Elem[operand2, e*2, esize]);
      element4 = Elem[operand1, e*2+1, esize];
  Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
  Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);
V[d] = result;
```
FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision

( ARMv8.2 )

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|--------------------|-----------------|-----------------|
| 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0 1 1 0 1 1 0 | Rn | Rd |

FCMLE <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

### Scalar single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|--------------------|-----------------|-----------------|
| 0 1 1 1 1 1 1 1 0 1 sz 1 0 0 0 0 0 1 1 0 1 1 0 | Rn | Rd |

FCMLE <V<d>, <V<n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;
Vector half precision
(Armv8.2)

FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Vector single-precision and double-precision

FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
  when '00' comparison = CompareOp_GT;
  when '01' comparison = CompareOp_GE;
  when '10' comparison = CompareOp_EQ;
  when '11' comparison = CompareOp_LE;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”: 
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:<T>”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<V_n>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \(V[n]\);
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```
**FCMLT (zero)**

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

**Scalar half precision**

**(Armv8.2)**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

```
FCMLT <Hd>, <Hn>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

**Scalar single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

```
FCMLT <V><d>, <V><n>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

**Vector half precision**

**(Armv8.2)**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
</tr>
</tbody>
</table>

```
FCMLT (zero)
```
FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;

Assembler Symbols

<Hz> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
V[d] = result;
```

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the \textit{PSTATE}.\{N, Z, C, V\} flags. This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer n = \textit{UInt}(Rn);
integer m = \textit{UInt}(Rm);    // ignored when \textit{opc}<0> == '1'

integer datasize;
\textbf{case ftype of}
  \textbf{when '00'} datasize = 32;
  \textbf{when '01'} datasize = 64;
  \textbf{when '10'} UNDEFINED;
  \textbf{when '11'}
    \textbf{if \textit{HaveFP16Ext}()} then
      datasize = 16;
    \textbf{else}
      UNDEFINED;

boolean signal_all_nans = (\textit{opc}<1> == '1');
boolean cmp_with_zero = (\textit{opc}<0> == '1');
```

### Half-precision (ftype == 11 && opc == 00)
\footnotesize (Armv8.2)

\texttt{FCMP} <\texttt{Hn}>, <\texttt{Hm}>

### Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 01)
\footnotesize (Armv8.2)

\texttt{FCMP} <\texttt{Hn}>, #0.0

### Single-precision (ftype == 00 && opc == 00)

\texttt{FCMP} <\texttt{Sn}>, <\texttt{Sm}>

### Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 01)

\texttt{FCMP} <\texttt{Sn}>, #0.0

### Double-precision (ftype == 01 && opc == 00)

\texttt{FCMP} <\texttt{Dn}>, <\texttt{Dm}>

### Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 01)

\texttt{FCMP} <\texttt{Dn}>, #0.0
Assembler Symbols

\(<Dn>\) For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

\(<Dm>\) Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

\(<Hn>\) For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

\(<Hm>\) Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

\(<Sn>\) For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

\(<Sm>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();

\(\text{bits(datasize) operand1} = V[n];\)
\(\text{bits(datasize) operand2};\)

\(\text{operand2} = \text{if cmp_with_zero then FPUzero('0') else V[m]};\)

\(\text{PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);}\)

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of \(<, ==, >\) or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the \(\text{PSTATE}\) condition flags to \(N=0, Z=0, C=1,\) and \(V=1.\)
FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the \text{PSTATE}.\{N, Z, C, V\} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in \text{FPCR}, the exception results in either a flag being set in \text{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \text{CPACR\_EL1}, \text{CPTR\_EL2}, and \text{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 1 1 0 ftype 1 Rm 0 0 1 0 0 0 Rn 1 x 0 0 0 opc
\end{verbatim}

**Half-precision (ftype == 11 && opc == 10)**

(Armv8.2)

\texttt{FCMPE <Hn>, <Hm>}

**Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 11)**

(Armv8.2)

\texttt{FCMPE <Hn>, #0.0}

**Single-precision (ftype == 00 && opc == 10)**

\texttt{FCMPE <Sn>, <Sm>}

**Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 11)**

\texttt{FCMPE <Sn>, #0.0}

**Double-precision (ftype == 01 && opc == 10)**

\texttt{FCMPE <Dn>, <Dm>}

**Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 11)**

\texttt{FCMPE <Dn>, #0.0}

```plaintext
integer n = UInt(Rn); integer m = UInt(Rm); // ignored when opc<0> == '1'
integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
data size = 16;
    else
      UNDEFINED;
    end
boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

FCMPE
Assembler Symbols

<Dn> For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Hn> For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<Sn> For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = if cmp_with_zero then FPZero('0') else V[m];
PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the PSTATE condition flags to N=0, Z=0, C=1, and V=1.
FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 0 ftype 1 Rm cond 1 1 Rn Rd</td>
</tr>
</tbody>
</table>

**Half-precision \((\texttt{ftype} == 11)\)**  
(Armv8.2)

\[
\text{FCSEL <Hd>, <Hn>, <Hm>, <cond>}
\]

**Single-precision \((\texttt{ftype} == 00)\)**

\[
\text{FCSEL <Sd>, <Sn>, <Sm>, <cond>}
\]

**Double-precision \((\texttt{ftype} == 01)\)**

\[
\text{FCSEL <Dd>, <Dn>, <Dm>, <cond>}
\]

\[
\begin{align*}
\text{integer } d &= \texttt{UInt}(Rd); \\
\text{integer } n &= \texttt{UInt}(Rn); \\
\text{integer } m &= \texttt{UInt}(Rm); \\
\text{integer } \text{datasize}; \\
\text{case } \texttt{ftype} \text{ of} \\
&\quad \text{when } '00' \text{ datasize } = 32; \\
&\quad \text{when } '01' \text{ datasize } = 64; \\
&\quad \text{when } '10' \text{ UNDEFINED}; \\
&\quad \text{when } '11' \\
&\quad \quad \text{if } \texttt{HaveFP16Ext}() \text{ then} \\
&\quad \quad \quad \text{datasize } = 16; \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{UNDEFINED};
\end{align*}
\]

**Assembler Symbols**

- \texttt{<Dd>} Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Dn>} Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Dm>} Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<Hd>} Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Hn>} Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Hm>} Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<Sn>} Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Sm>} Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Sm>} Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<cond>} Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) result;
result = if ConditionHolds(cond) then V[n] else V[m];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
FCVT

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the \texttt{FPCR} and writes the result to the SIMD&FP destination register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>ftype</th>
<th>opc</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>

\textbf{Half-precision to single-precision (ftype == 11 && opc == 00)}

\texttt{FCVT <Sd>, <Hn>}

\textbf{Half-precision to double-precision (ftype == 11 && opc == 01)}

\texttt{FCVT <Dd>, <Hn>}

\textbf{Single-precision to half-precision (ftype == 00 && opc == 11)}

\texttt{FCVT <Hd>, <Sn>}

\textbf{Single-precision to double-precision (ftype == 00 && opc == 01)}

\texttt{FCVT <Dd>, <Sn>}

\textbf{Double-precision to half-precision (ftype == 01 && opc == 11)}

\texttt{FCVT <Hd>, <Dn>}

\textbf{Double-precision to single-precision (ftype == 01 && opc == 00)}

\texttt{FCVT <Sd>, <Dn>}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer srcsize;
integer dstsize;
if ftype == opc then UNDEFINED;

case ftype of
  when '00' srcsize = 32;
  when '01' srcsize = 64;
  when '10' UNDEFINED;
  when '11' srcsize = 16;
  case opc of
    when '00' dstsize = 32;
    when '01' dstsize = 64;
    when '10' UNDEFINED;
    when '11' dstsize = 16;
\end{verbatim}

\textbf{Assembler Symbols}

\texttt{<Dd>} Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<Hd>} Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<Sn>} Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(srcsize) operand = V[n];
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bitalign(128) result = if merge then V[d] else Zeros();
Elem[result, 0, dstsize] = FPConvert(operand, fpcr);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1, CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

**Scalar half precision**

(*Armv8.2*)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 0 1 1 1 0 0 | 1 1 1 0 0 1 1 0 0 | Rn | Rd
```

```
UFCVTAS <Hd>, <Hn>
   if !HaveFP16Ext() then UNDEFINED;
   integer d = UInt(Rd);
   integer n = UInt(Rn);
   integer esize = 16;
   integer datasize = esize;
   integer elements = 1;

   FPRounding rounding = FPRounding_TIEAWAY;
   boolean unsigned = (U == '1');
```

**Scalar single-precision and double-precision**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 0 1 1 1 0 0 | sz | 1 0 0 0 0 | 1 1 1 0 0 1 0 | Rn | Rd
```

```
UFCVTAS <V><d>, <V><n>
   integer d = UInt(Rd);
   integer n = UInt(Rn);
   integer esize = 32 << UInt(sz);
   integer datasize = esize;
   integer elements = 1;

   FPRounding rounding = FPRounding_TIEAWAY;
   boolean unsigned = (U == '1');
```

**Vector half precision**

(*Armv8.2*)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 0 | 0 1 1 1 0 0 | 1 1 1 0 0 1 1 0 0 1 0 | Rn | Rd
```

```
UFCVTAS (vector)
```
FCVTAS <Vd><T>, <Vn><T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|     | Q  | 0  | 1  | 1  | 1  | 0  | 0  | s  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rd |
|     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|     | U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FCVTAS <Vd><T>, <Vn><T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Assembler Symbols

<Hd>  Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>  Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V>   Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
<d>  Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n>  Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>  For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

V[d] = result;
FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf |0| 0| 1| 1| 1| 1| 0| ftype |1| 0| 0| 1| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| Rn | Rd
```

rmode opcode

Half-precision to 32-bit (sf == 0 && ftype == 11)  
(Armv8.2)

```
FCVTAS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(Armv8.2)

```
FCVTAS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTAS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTAS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTAS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTAS <Xd>, <Dn>
```

```plaintext
total = integer d = UInt(Rd);
total n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
```

FCVTAS (scalar)
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
FPCRTypefpcr = FPCR[];
bitts(fltsize) fltval;
bitts(intsize) intval;
fltval = \text{V}[n];
intval = \text{FPToFixed}(fltval, 0, FALSE, fpcr, \text{FPRounding\_TIEAWAY});
X[d] = intval;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

**Scalar half precision**

(Armv8.2)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| U | Rn | Rd |
```

```
FCVTAU <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

**Scalar single-precision and double-precision**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| U | Rn | Rd |
```

```
FCVTAU <V<d>, <V<n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

**Vector half precision**

(Armv8.2)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| U | Rn | Rd |
```
FCVTAU <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FCVTAU <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th></th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf | 0 | 0 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
```

rmode  opcode

Half-precision to 32-bit (sf == 0 && ftype == 11)  
(Armv8.2)

FCVTAU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(Armv8.2)

FCVTAU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)  

FCVTAU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)  

FCVTAU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)  

FCVTAU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)  

FCVTAU <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;integer fltsize;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
```
**Assembler Symbols**

- `<Wd>` Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Xd>` Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;
fltval = Y[n];
intval = FPtoFixed(fltval, 0, TRUE, fpcr, FPRounding_TIEAWAY);
X[d] = intval;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTL, FCVTL2

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the FPCR, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the FCVTL2 variant operates on the elements in the top 64 bits of the source register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Binary Representation](image)

**Assembler Symbols**

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

|(sz:Q) <Tb> |
|----------|---------------|
| 0 0      | 4H             |
| 0 1      | 8H             |
| 1 0      | 2S             |
| 1 1      | 4S             |

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

```
FCVTMS <Hd>, <Hn>
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

```
FCVTMS <V>d>, <V>n>
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision
(Armv8.2)

```
FCVTMS (vector)
Page 737
```
FCVTMS <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  |
| U  | o2 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

FCVTMS <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \texttt{V}[n];

bits(esize) element;
\texttt{FPCRTYPE} fpcr = FPCR[];
boolean merge = elements == 1 && \texttt{IsMerging}(fpcr);
bших(128) result = if merge then \texttt{V}[d] else \texttt{Zeros}();

for e = 0 to elements-1
  element = \texttt{Elem}(operand, e, esize);
  \texttt{Elem}(result, e, esize) = \texttt{FPToFixed}(element, 0, unsigned, fpcr, rounding);
\texttt{V}[d] = result;
FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Half-precision to 32-bit (sf == 0 && ftype == 11)  
(Armv8.2)

FCVTMS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(Armv8.2)

FCVTMS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTMS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTMS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTMS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTMS <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;
```
**FCVTMU (vector)**

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

**Scalar half precision**

(Armv8.2)

```
|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------|----------|
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |
|   |   | U | o2     | 01       |

FCVTMU <Hd>, <Hn>
```

|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------|----------|
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |
|   |   | U | o2     | 01       |

FCVTMU <V><d>, <V><n>

|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------|----------|
| 0 | 0 | 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | Rn | Rd |
|   |   | U | o2     | 01       |
```

**Scalar single-precision and double-precision**

```
```

**Vector half precision**

(Armv8.2)

```
```

**Vector single-precision and double-precision**

```
```
FCVTMU <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer data-size = if Q == '1' then 128 else 64;
integer elements = data-size DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

**Vector single-precision and double-precision**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Assembler Symbols**

<HD> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<HN> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTypedef pcr = FPCR[];
boolean merge = elements == 1 && IsMerging(pcr);
bets(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, pcr, rounding);
V[d] = result;
```
FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Half-precision to 32-bit (sf == 0 & ftype == 11) (Armv8.2)

FCVTMU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 & ftype == 11) (Armv8.2)

FCVTMU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 & ftype == 00)

FCVTMU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 & ftype == 00)

FCVTMU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 & ftype == 01)

FCVTMU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 & ftype == 01)

FCVTMU <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    rounding = FPDecteRounding(rmode);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypelfpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding);
X[d] = intval;
FCVTN, FCVTN2

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the FPCR.

The FCVTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
0   Q   0   0   1   1   1   0   0   0   1   0   1   1   0   1   0   Rn   Rd
```

FCVTN(2) <Vd>.<Tb>, <Vn>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>absent</td>
</tr>
<tr>
<td>1</td>
<td>present</td>
</tr>
</tbody>
</table>
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

```
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “sz”:

```
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
```

**Operation**

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR[]);

Vpart[d, part] = result;
FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

**Scalar half precision**

(Armv8.2)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  1  1  1  1  0  0  1  1  1  0  0  1  1  0  1  0  1  0  Rn  Rd

FCVTNS <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

**Scalar single-precision and double-precision**

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  1  1  1  1  0  0  sz  1  0  0  0  0  1  1  0  1  0  1  0  Rn  Rd

FCVTNS <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

**Vector half precision**

(Armv8.2)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  1  1  1  0  0  1  1  1  1  0  0  1  1  0  1  0  1  0  Rn  Rd

FCVTNS (vector)
```
FCVTNS $\langle Vd \rangle . \langle T \rangle , \langle Vn \rangle . \langle T \rangle$

if !HaveFP16Ext() then UNDEFINED;

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;

integer $esize = 16$;
integer $datasize = \text{if } Q == \text{'1'} \text{ then } 128 \text{ else } 64$;
integer $elements = \text{datasize} \text{ DIV } esize$;

FPRounding $\text{rounding} = \text{FPDecodeRounding}(o1:o2)$;
boolean $\text{unsigned} = (U == \text{'1'})$;

**Vector single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   |   |   |   | sz | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |   |   |   |   |
| 0 |   |   |   |   | U | 0 | 2 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FCVTNS $\langle Vd \rangle . \langle T \rangle , \langle Vn \rangle . \langle T \rangle$

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;

if $sz:Q == \text{'10'}$ then UNDEFINED;
integer $esize = 32 \text{ << } \text{UInt}(sz)$;
integer $datasize = \text{if } Q == \text{'1'} \text{ then } 128 \text{ else } 64$;
integer $elements = \text{datasize} \text{ DIV } esize$;

FPRounding $\text{rounding} = \text{FPDecodeRounding}(o1:o2)$;
boolean $\text{unsigned} = (U == \text{'1'})$;

**Assembler Symbols**

$<Hd>$ Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Hn>$ Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

$<V>$ Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>$&lt;V&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

$<d>$ Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

$<n>$ Is the number of the SIMD&FP source register, encoded in the "Rn" field.

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\textbf{CheckFPAdvSIMDEnabled64}();
\textbf{bits}(\texttt{datasize}) \texttt{operand} = \textit{V}[n];

\textbf{bits}(\texttt{esize}) \texttt{element};
\textbf{FPCRTYPE} \texttt{fpcr} = \text{FPCR}[];
\textbf{boolean} \text{merge} = \text{elements} == 1 \&\& \text{IsMerging}(\texttt{fpcr});
\textbf{bits}(128) \texttt{result} = \text{if} \text{merge} \text{then} \textit{V}[d] \text{else} \textit{Zeros}();

\textbf{for} e = 0 \text{to} \text{elements-1}
\text{element} = \texttt{Elem}[	exttt{operand}, e, \texttt{esize}];
\texttt{Elem}[	exttt{result}, e, \texttt{esize}] = \textbf{FPToFixed}(	exttt{element}, 0, \text{unsigned}, \texttt{fpcr}, \text{rounding});
\textit{V}[d] = \texttt{result};
FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
```

sf: Source float
ftype: Ftype
rmode: Rmode
opcode: Opcode
Rn: 32-bit register
Rd: 64-bit register
Half-precision to 32-bit (sf == 0 \&\& ftype == 11) (Armv8.2)

FCVTNS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 \&\& ftype == 11) (Armv8.2)

FCVTNS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 \&\& ftype == 00)

FCVTNS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 \&\& ftype == 00)

FCVTNS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 \&\& ftype == 01)

FCVTNS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 \&\& ftype == 01)

FCVTNS <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = FPDecodeRounding(rmode);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\texttt{CheckFPAdvSIMEnabled64();}

\texttt{FPCRTyp}e \ fpcr = FPCR[];
\texttt{bits(fltsize)} fltval;
\texttt{bits(intsize)} intval;

fltval = \texttt{V}[n];
intval = \texttt{FPToFixed}(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;
FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

\[
\begin{array}{cccccccccccccccccccc}
\hline
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & Rn & Rd \\
\end{array}
\]

FCVTNU <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

\[
\begin{array}{cccccccccccccccccccc}
\hline
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & sz & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & Rn & Rd \\
\end{array}
\]

FCVTNU <V>d>, <V>n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision
(Armv8.2)

\[
\begin{array}{cccccccccccccccccccc}
\hline
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & Rn & Rd \\
\end{array}
\]
FCVTNU <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
Half-precision to 32-bit (sf == 0 && ftype == 11)  
(Armv8.2)

FCVTNU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(Armv8.2)

FCVTNU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTNU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTNU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTNU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTNU <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
rounding = FPDecodeRounding(rmode);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = Y[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding);
X[d] = intval;
```
FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

<table>
<thead>
<tr>
<th>0 1</th>
<th>0 1 1 1 0</th>
<th>0 1</th>
<th>0 1 0 1 0</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td>Rn</td>
<td></td>
</tr>
</tbody>
</table>

FCVTPS <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>0 1</th>
<th>0 1 1 1 0</th>
<th>0 1</th>
<th>sz</th>
<th>1 0 0 0</th>
<th>1 1 0 1</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FCVTPS <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision
(Armv8.2)

<table>
<thead>
<tr>
<th>0 0 1 1 1 0</th>
<th>0 1</th>
<th>0 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
</tr>
</tbody>
</table>

FCVTPS (vector)
FCVTPS <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|   | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
| U | 02 |     |

FCVTPS <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \( V[n] \);

bits(esize) element;
FPCRTyp fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then \( V[d] \) else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);

\( V[d] = \) result;
FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
+31 +30 +29 +28 +27 +26 +25 +24 +23 +22 +21 +20 +19 +18 +17 +16 +15 +14 +13 +12 +11 +10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
| sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
```

opcode  rmode
Half-precision to 32-bit (sf == 0 && ftype == 11) (Armv8.2)

FCVTPS <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11) (Armv8.2)

FCVTPS <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTPS <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTPS <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTPS <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTPS <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDetectRounding(rmode);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTPU (vector)**

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPSCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

**Scalar half precision**

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 3 2 1 0 |
|------------------|------------------|
| 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 0 1 0 0 1 0 0 1 1 0 1 0 0 1 1 0 |
|                  |                  |
| Rn               | Rd               |
| 02               | 01               |

FCVTPU <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPDecoding rounding = FPDecoding(o1:o2);
boolean unsigned = (U == '1');

**Scalar single-precision and double-precision**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 3 2 1 0 |
|------------------|------------------|
| 0 1 1 1 1 1 1 0 1 sz 1 0 0 0 0 1 1 0 1 0 0 1 0 0 1 1 0 1 0 |
|                  |                  |
| Rn               | Rd               |
| 02               | 01               |

FCVTPU <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPDecoding rounding = FPDecoding(o1:o2);
boolean unsigned = (U == '1');

**Vector half precision**

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 3 2 1 0 |
|------------------|------------------|
| 0 1 1 1 1 1 0 1 1 1 1 0 0 1 1 0 1 0 0 1 1 0 1 0 0 1 1 0 |
|                  |                  |
| Rn               | Rd               |
| 02               | 01               |

FCVTPU (vector)
FCVTPU <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
|                   | sz              | Q               |
| U                | 02              | Rd              |
|                  | 0               | sz              |
| o1               | 10              | 0               |
|                  | 00              | 0               |
|                  | 11              | 0               |
|                  | 10              | 0               |
|                  | 01              | 0               |

FCVTPU <Vd>,<T>, <Vn>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

`CheckFPAdvSIMDEnabled64();`

`bits(datasize) operand = V[n];`

`bits(esize) element;`

`FPCRTypedefpcr = FPCR[];`

`boolean merge = elements == 1 && IsMerging(fpcr);`

`bits(128) result = if merge then V[d] else Zeros();`

`for e = 0 to elements-1`
  `element = Elem[operand, e, esize];`
  `Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);`

`V[d] = result;`
FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf | 0 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Rn | Rd
rmode | opcode
\end{verbatim}
Half-precision to 32-bit (sf == 0 && ftype == 11)  
(Armv8.2)

FCVTPU <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11)  
(Armv8.2)

FCVTPU <Xd>, <Hn>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTPU <Wd>, <Sn>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTPU <Xd>, <Sn>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTPU <Wd>, <Dn>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTPU <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  rounding = FPDcodeRounding(rmode);

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\text{FPCRTyp}e \ fpcr = \text{FPCR}[];
\]

\[
\text{bits}(\text{fltsize}) \ \text{fltval};
\]

\[
\text{bits}(\text{intsize}) \ \text{intval};
\]

\[
\text{fltval} = V[n];
\]

\[
\text{intval} = \text{FPToFixed}(\text{fltval}, 0, \text{TRUE}, \text{fpcr}, \text{rounding});
\]

\[
X[d] = \text{intval};
\]
FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--------------------------|
| 0 1 1 1 1 1 1 0 0 | sz 1 0 0 0 0 1 0 1 1 0 1 0 | Rd |

FCVTXN `<Vb><d>`, `<Va><n>`

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;

### Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--------------------------|
| 0 Q 1 1 1 1 0 0 | sz 1 0 0 0 0 1 0 1 1 0 1 0 | Rd |

FCVTXN{2} `<Vd>.<Tb>`, `<Vn>.<Ta>`

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);

### Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>0 [absent]</th>
<th>1 [present]</th>
</tr>
</thead>
</table>

Q 2

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Tb> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(2*datasize) operand = V[n];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 & IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], fpcr, FPRounding_ODD);
if merge then
    V[d] = result;
ext Else
    Vpart[d, part] = Elem[result, 0, datasize];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTZS (vector, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR EL1}, \textit{CPTR EL2}, and \textit{CPTR EL3} registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \textit{Scalar} and \textit{Vector}

**Scalar**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  |    |    |    |    | 1  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | Rn | Rd |

FCVTZS <V>d>, <V>n>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh == '000x' || (immh == '001x' \&\& !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

**Vector**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  |    |    |    |    | 1  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | Rn | Rd |

FCVTZS <Vd>.,<T>, <Vn>.<T>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' \&\& !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

**Assembler Symbols**

\textit{<V>} Is a width specifier, encoded in “immh”:

```

FCVTZS (vector, fixed-point) Page 777
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>011x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
</tbody>
</table>

<fn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>fbits</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-Uint(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>fbits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE先进SIMD修改立即</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-Uint(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-Uint(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvanceSIMDEnabled64();
bits(datosize) operand = V[n];
bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
   element = Elem[operand, e, esize];
   Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding);
V[d] = result;
FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(ARMv8.2)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
     0  1  1  1  1  0  1  1  1  1  0  0  1  1  0  1  1  0
Rn   o2
```

```
FCVTZS <Hd>, <Hn>
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
     0  1  1  1  1  0  1  sz  1  0  0  0  0  1  1  0  1  1  0
Rn   o2
```

```
FCVTZS <V>d>, <V>n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

(ARMv8.2)

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
     0  1  1  1  1  0  1  1  1  1  0  0  1  1  0  1  1  0
Rn   o2
```

```
FCVTZS (vector, integer) Page 779
FCVTZS <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

type integer d = UInt(Rd);
type integer n = UInt(Rn);

type integer esize = 16;
type integer datasize = if Q == '1' then 128 else 64;
type integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | sz | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
| U  | o2 |
| o1 |

FCVTZS <Vd>.<T>, <Vn>.<T>

int d = UInt(Rd);
int n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
type integer esize = 32 << UInt(sz);
type integer datasize = if Q == '1' then 128 else 64;
type integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```
FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 0 | 1 | 1 | 0 | 0 | scale | Rn | Rd
      <Wd>, <Hn>, #<fbits>
```

**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(Armv8.2)

FCVTZS <Wd>, <Hn>, #<fbits>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(Armv8.2)

FCVTZS <Xd>, <Hn>, #<fbits>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTZS <Wd>, <Sn>, #<fbits>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTZS <Xd>, <Sn>, #<fbits>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTZS <Wd>, <Dn>, #<fbits>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTZS <Xd>, <Dn>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
```
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".
For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, frachbits, FALSE, fpcr, FPRounding_ZERO);
X[d] = intval;
FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| sf | 0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd | rmode | opcode |
**Half-precision to 32-bit (sf == 0 && ftype == 11)**  
(Armv8.2)

FCVTZS <Wd>, <Hn>

**Half-precision to 64-bit (sf == 1 && ftype == 11)**  
(Armv8.2)

FCVTZS <Xd>, <Hn>

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

FCVTZS <Wd>, <Sn>

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

FCVTZS <Xd>, <Sn>

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

FCVTZS <Wd>, <Dn>

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

FCVTZS <Xd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    rounding = FPDecodeRounding(rmode);

**Assembler Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypelfcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = Y[n];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding);
X[d] = intval;
FCVTZU (vector, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------------------|
| U | != 0000 | immh | Rn | Rd |

FCVTZU <V><d>, <V><n>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer elements = 1;
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

**Vector**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------------------|
| U | != 0000 | immh | Rn | Rd |

FCVTZU <Vd>,<T>, <Vn>,<T>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer elements = datasuresize DIV esize;
integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

**Assembler Symbols**

<V> Is a width specifier, encoded in "immh":

integer fracbits = (esize * 2) - UInt(immh:immb);

boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

<fbits> For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding);
V[d] = result;
FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FCVTZU <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FCVTZU <V>d>, <V>n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 >> UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');

Vector half precision
(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FCVTZU (vector, integer)
FCVTZU \(<V_d>\.T>, \ <V_n>\.T>

if \(!\text{HaveFP16Ext}()\) then UNDEFINED;

integer \(d = \text{UInt}(R_d)\);
integer \(n = \text{UInt}(R_n)\);

integer esize = 16;
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

\(\text{FPRounding} \: \text{rounding} = \text{FPDecodeRounding}(o_1:o_2)\);
boolean unsigned = (\(U == '1'\));

**Vector single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 0  | 1  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |
| U  |    |    |    |    |    |    | o2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| D  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FCVTZU \(<V_d>\.T>, \ <V_n>\.T>

integer \(d = \text{UInt}(R_d)\);
integer \(n = \text{UInt}(R_n)\);

if \(sz:Q == '10'\) then UNDEFINED;
integer esize = 32 << \(\text{UInt}(sz)\);
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

\(\text{FPRounding} \: \text{rounding} = \text{FPDecodeRounding}(o_1:o_2)\);
boolean unsigned = (\(U == '1'\));

**Assembler Symbols**

\(<H_d>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<H_n>\) Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

\(<V>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<V_n>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

bits(esize) element;
FPCRTypetype fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
   element = Elem[operand, e, esize];
   Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding);
V[d] = result;
FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
sf 0 0 1 1 1 1 0 ftype 0 1 1 0 0 1 scale Rd
            Rn
            mode opcode

Half-precision to 32-bit (sf == 0 && ftype == 11)
(Armv8.2)

FCVTZU <Wd>, <Hn>, #<fbits>

Half-precision to 64-bit (sf == 1 && ftype == 11)
(Armv8.2)

FCVTZU <Xd>, <Hn>, #<fbits>

Single-precision to 32-bit (sf == 0 && ftype == 00)

FCVTZU <Wd>, <Sn>, #<fbits>

Single-precision to 64-bit (sf == 1 && ftype == 00)

FCVTZU <Xd>, <Sn>, #<fbits>

Double-precision to 32-bit (sf == 0 && ftype == 01)

FCVTZU <Wd>, <Dn>, #<fbits>

Double-precision to 64-bit (sf == 1 && ftype == 01)

FCVTZU <Xd>, <Dn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
    if sf == '0' && scale<5> == '0' then UNDEFINED;
    integer fracbits = 64 - UInt(scale);
Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits> For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale".
For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```c
CheckFPAdvSIMDEnabled64();

FPCRTYPE fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, frachbits, TRUE, fpcr, FPRounding_ZERO);
X[d] = intval;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCVTZU (scalar, integer)**

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td>ftype</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

mode opcode
### Half-precision to 32-bit (sf == 0 && ftype == 11)
(Armv8.2)

```
FCVTZU <Wd>, <Hn>
```

### Half-precision to 64-bit (sf == 1 && ftype == 11)
(Armv8.2)

```
FCVTZU <Xd>, <Hn>
```

### Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTZU <Wd>, <Sn>
```

### Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTZU <Xd>, <Sn>
```

### Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTZU <Wd>, <Dn>
```

### Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTZU <Xd>, <Dn>
```

integer \( d \) = \text{UInt}(\text{Rd});

integer \( n \) = \text{UInt}(\text{Rn});

integer \( \text{intsize} \) = \text{if} \( \text{sf} == '1' \) \text{then} 64 \text{else} 32;

ingen \( \text{fltsize} \);

\text{FPRounding} \ \text{rounding};

\text{case} \ \text{ftype} \ \text{of}
\quad \text{when} \ '00'
\quad \quad \text{fltsize} = 32;
\quad \text{when} \ '01'
\quad \quad \text{fltsize} = 64;
\quad \text{when} \ '10'
\quad \quad \text{UNDEFINED};
\quad \text{when} \ '11'
\quad \quad \text{if} \ \text{HaveFP16Ext}() \ \text{then}
\quad \quad \text{fltsize} = 16;
\quad \text{else}
\quad \quad \text{UNDEFINED};
\text{rounding} = \text{FPDecodeRounding}(\text{rmode});

### Assembler Symbols

- **<Wd>** Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Xd>** Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- **<Sn>** Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Hn>** Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Dn>** Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMEnabled64();

FPCRTypelfcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding);
X[d] = intval;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   |   |   |   |   |   |   |   |   |   |   | Q  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   | 0  | 0  | 1  | 1  | 1  |   | 1  |   | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   |   |   |   |   |   |   |   |   |   |   | Q  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   | 0  | 0  | 1  | 1  | 1  | 1  |   | 1  |   | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
<td></td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FDIV (scalar)**

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | Rm  | 0  | 0  | 0  | 1  | 1  | 0  | Rd
```

**Half-precision (ftype == 11)**

(Armv8.2)

```
FDIV <Hd>, <Hn>, <Hm>
```

**Single-precision (ftype == 00)**

```
FDIV <Sd>, <Sn>, <Sm>
```

**Double-precision (ftype == 01)**

```
FDIV <Dd>, <Dn>, <Dm>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
(case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
end)

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPDiv(operand1, operand2, FPCR[]);

V[d] = result;
Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be represented as a signed 32-bit integer, then the result is the integer modulo $2^{32}$, as held in a 32-bit signed integer. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Double-precision to 32-bit**

(Armv8.3)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |

sf ftype rmode opcode

FJCVTZS \(<Wd>, <Dn>\)

integer d = UInt(Rd);
integer n = UInt(Rn);

if !HaveFJCVTZSExt() then UNDEFINED;

**Assembler Symbols**

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPAdvSIMDEnabled64();

FPCRTRef fpcr = FPCR[];
bits(64) fltval;
bits(32) intval;

bit Z;
fltval = V[n];
(intval, Z) = FPtoFixedJS(fltval, fpcr, TRUE);
PSTATE.<N,Z,C,V> = '0':Z:'00';
X[d] = intval;


### FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 1  | ftype | 0  | Rm | 0  | Ra | 0  | Rn | 0  | Rd | 0  |
| 01 | 00 |

**Half-precision (ftype == 11)**

(ARMv8.2)

\[
\text{FMADD } <Hd>, <Hn>, <Hm>, <Ha> 
\]

**Single-precision (ftype == 00)**

\[
\text{FMADD } <Sd>, <Sn>, <Sm>, <Sa> 
\]

**Double-precision (ftype == 01)**

\[
\text{FMADD } <Dd>, <Dn>, <Dm>, <Da> 
\]

```plaintext
def integer d = UInt(Rd);
def integer a = UInt(Ra);
def integer n = UInt(Rn);
def integer m = UInt(Rm);

def integer esize;
def case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

**Assembler Symbols**

- `<Dd>` is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Dm>` is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Da>` is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- `<Hd>` is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Hm>` is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Ha>` is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

### Operation

```cpp
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 0       | 0       | 0       | 1       | 1       | 1       | 0       | 0       | 1       | 0       | Rm      | 0       | 0       | 1       | 1       | 0       | 1       | Rn      | Rd      |
| U       | o1      |

FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Single-precision and double-precision**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 0       | 0       | 0       | 1       | 1       | 1       | 0       | 0       | sz      | 1       | Rm      | 1       | 1       | 1       | 1       | 0       | 1       | Rn      | Rd      |
| U       | o1      |

FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Assembler Symbols**

<Vd> is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vn>**  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>**  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1   | Rm | 0  | 1  | 0  | 0  | 1  | 0  | Rn  |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Half-precision (ftype == 11)**

(Armv8.2)

FMAX <Hd>, <Hn>, <Hm>

**Single-precision (ftype == 00)**

FMAX <Sd>, <Sn>, <Sm>

**Double-precision (ftype == 01)**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
b bits(esize) operand1 = V[n];
b bits(esize) operand2 = V[m];

FPCRTyp e fp cr = FPCR[];
boolean merge = IsMerging(fp cr);
b bits(128) result = if merge then V[n] else Zeros();

E le m[ result, 0, esize] = FPMax(operand1, operand2, fpcr);
V[d] = result;
\end{verbatim}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNM (vector)

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  |   |   | Rm | 0  | 0  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| U  | a  |
```

FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

**Single-precision and double-precision**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| U  | 01 |
```

FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (01 == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:  

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bounds(datasize) operand1 = V[n];
bounds(datasize) operand2 = V[m];
bounds(datasize) result;
bounds(2*datasize) concat = operand2:operand1;
bounds(esize) element1;
bounds(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem(concat, 2*e, esize);
        element2 = Elem(concat, (2*e)+1, esize);
    else
        element1 = Elem(operand1, e, esize);
        element2 = Elem(operand2, e, esize);
    if minimum then
        Elem(result, e, esize) = FPMinNum(element1, element2, FPCR[]);
    else
        Elem(result, e, esize) = FPMaxNum(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 1  | 0  | ftype | 1  | Rm  | 0  | 1  | 1  | 0  | 1  | Rd  |

### Half-precision (ftype == 11)
(Armv8.2)

FMAXNM <Hd>, <Hn>, <Hm>

### Single-precision (ftype == 00)

FMAXNM <Sd>, <Sn>, <Sm>

### Double-precision (ftype == 01)

FMAXNM <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
when '00' esize = 32;
when '01' esize = 64;
when '10' UNDEFINED;
when '11' if HaveFP16Ext() then esize = 16;
else UNDEFINED;

Assembler Symbols

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Dn>** Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Dm>** Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Hm>** Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Sn>** Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Sm>** Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);
V[d] = result;
FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(Armv8.2)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Rn | Rd |
| 0 | 1 |   |   |   |   |   |   | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Rn | Rd |

\[\text{FMAXNMP} \text{ <V><d>}, \text{ <Vn>}.<T>}\]

if !HaveFP16Ext() then UNDEFINED;

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

integer esize = 16;
integer datasize = 32;

### Single-precision and double-precision

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Rn | Rd |
| 0 | 1 |   |   |   |   |   |   | 0 | sz |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Rn | Rd |

FMAXNMP \(\text{<V><d>}, \text{ <Vn>}.<T>\)

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

integer esize = 32;
integer datasize = 64;

### Assembler Symbols

\(<V>\) For the half-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) For the half-precision variant: is the source arrangement specifier, encoded in “sz”:
For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
| U  | a |

FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| U  | ol |

FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (ol == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{Vm}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to FMAX (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | Q | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd |
| 01                              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FMAXNMV <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;

### Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd |
| 01                              |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FMAXNMV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;

### Assembler Symbols

**<V>**

For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**<d>**

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<Vn>**

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<T>**

For the half-precision variant: is an arrangement specifier, encoded in "Q";
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

01 sz

Rn Rd

FMAXP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;

Single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

01 sz

Rn Rd

FMAXP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of
the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair
of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and
writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.
This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in
either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point
exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|
| 0 Q 1 0 1 1 1 0 0 | 1 0 Rm | 0 0 1 1 0 1 Rn | Rd |
| U o1              |      |              |     |

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|-----------------|
| 0 Q 1 0 1 1 1 0 0 | sz 1 Rm | 1 1 1 1 0 1 Rn | Rd |
| U o1              |      |              |     |

FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

Page 820
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{Vm}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bis(esize) element1;
bis(esize) element2;

for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];

  if minimum then
    Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FMAXV <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d =(UInt(Rd));
integer n = (UInt(Rn));

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FMAXV <V><d>, <Vn>.<T>

integer d = (UInt(Rd));
integer n = (UInt(Rn));

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << (UInt(sz));
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.
For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMIN (vector)**

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision.

### Half-precision (Armv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

### Single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

### Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

---
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{Vn}> \quad \text{Is the name of the first SIMD&FP source register, encoded in the "Rn" field.}

<\text{Vm}> \quad \text{Is the name of the second SIMD&FP source register, encoded in the "Rm" field.}

**Operation**

```c
CheckFPAdvSIMDEnabled64();

\text{bits(datasize)} \text{ operand1} = \text{V}\lbrack n\rbrack;  
\text{bits(datasize)} \text{ operand2} = \text{V}\lbrack m\rbrack;  
\text{bits(datasize)} \text{ result};  
\text{bits(2*datasize)} \text{ concat} = \text{operand2:operand1};  
\text{bits(esize)} \text{ element1};  
\text{bits(esize)} \text{ element2};

\text{for } e = 0 \text{ to elements-1}
    \begin{array}{l}
    \text{if pair then}\n    \quad \text{element1} = \text{Elem}\lbrack \text{concat}, 2\cdot e, \text{esize}\rbrack;  
    \quad \text{element2} = \text{Elem}\lbrack \text{concat}, (2\cdot e)+1, \text{esize}\rbrack;  
    \text{else}\n    \quad \text{element1} = \text{Elem}\lbrack \text{operand1}, e, \text{esize}\rbrack;  
    \quad \text{element2} = \text{Elem}\lbrack \text{operand2}, e, \text{esize}\rbrack;  
    \end{array}

    \text{if minimum then}\n    \begin{array}{l}
    \quad \text{Elem}\lbrack \text{result}, e, \text{esize}\rbrack = \text{FPMin}(\text{element1}, \text{element2}, \text{FPCR}[]);  
    \text{else}\n    \quad \text{Elem}\lbrack \text{result}, e, \text{esize}\rbrack = \text{FPMax}(\text{element1}, \text{element2}, \text{FPCR}[]);  
    \end{array}

\text{V}\lbrack d\rbrack = \text{result};
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMIN (scalar)**

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```plaintext
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | Rm | 0  | 1  | 0  | 1  | 1  | 0  | Rn | Rd |
```

**Half-precision (ftype == 11)**

(Armv8.2)

FMIN <Hd>, <Hn>, <Hm>

**Single-precision (ftype == 00)**

FMIN <Sd>, <Sn>, <Sm>

**Double-precision (ftype == 01)**

FMIN <Dd>, <Dn>, <Dm>

```plaintext
d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
switch ftype of
| case '00' esize = 32;
| case '01' esize = 64;
| case '10' UNDEFINED;
| case '11'
|   if HaveFP16Ext() then
|     esize = 16;
|   else
|     UNDEFINED;
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Floating-point Minimum (scalar)
Operation

CheckFPAdvSIMDEnabled64();
bias(esize) operand1 = V[n];
bias(esize) operand2 = V[m];

FPCRTypc fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bias(128) result = if merge then V[n] else Zeros();

Elem(result, 0, esize) = FPMin(operand1, operand2, fpcr);
V[d] = result;
FMINNM (vector)

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
| U  | a |

FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| U  | o1 |

FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

FMINNM (vector)
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{n}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNM (scalar)

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTER_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 0 ftype 1 Rm 0 1 1 1 1 0 Rn Rd
```

**Half-precision (ftype == 11)**

(Armv8.2)

FMINNM <Hd>, <Hn>, <Hm>

**Single-precision (ftype == 00)**

FMINNM <Sd>, <Sn>, <Sm>

**Double-precision (ftype == 01)**

FMINNM <Dd>, <Dn>, <Dm>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Sm>` Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTyp fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem[result, 0, esize] = FPMinNum(operand1, operand2, fpcr);
V[d] = result;
\end{verbatim}
FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd |
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

FMINNMP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = 32;

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd |
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

FMINNMP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

(Armv8.2)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
| U  | a |
```

FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');

**Single-precision and double-precision**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| U  | o1 |
```

FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if pair then
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
  else
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
  if minimum then
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FLOATING MINIMUM NUMBER ACROSS VECTOR (FMINNMV)

This instruction compares all the vector elements in the source SIMD&FP register and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to FMIN (scalar).

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision.

**Half-precision**

(ARMv8.2)

```plaintext
0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
```

**Single-precision and double-precision**

```plaintext
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
```

**Assembler Symbols**

- `<V>`: For the half-precision variant: is the destination width specifier, H.
- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- `<T>`: For the half-precision variant: is an arrangement specifier, encoded in "Q".
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
<td>1</td>
</tr>
</tbody>
</table>

Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\text{bits(datasize)} \text{ operand } = V[n];
V[d] = \text{Reduce(ReduceOp_FMINNUM, operand, esize, FALSE)};
\]

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 0 1 0 1 1 0 0 0 0 1 1 1 1 1 0 Rn Rd
```

FMINP <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n =.UInt(Rn);

integer esize = 16;
integer datasize = 32;

Single-precision and double-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 0 1 sz 1 1 0 0 0 0 0 1 1 1 1 1 0 Rn Rd
```

FMINP <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32;
integer datasize = 64;

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in "sz":

```
sz <V>
0 H
1 RESERVED
```

For the single-precision and double-precision variant: is the destination width specifier, encoded in "sz":

```
sz <V>
0 S
1 D
```

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in "sz":

```
sz <T>
0 2H
1 RESERVED
```
For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**Operation**

\[
\text{CheckFPAdvSIMDEnabled64}() ; \\
\text{bits(datasize) operand} = V[n] ; \\
V[d] = \text{Reduce(ReduceOp_FMIN, operand, esize)} ;
\]
FMINP (vector)

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision**

( Armv8.2 )

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (ol == '1');

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (ol == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

| 0 | 4H |
| 1 | 8H |

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vn>**  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>**  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

\[
\begin{array}{ccccccccccccccccccccccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & \quad \text{Rn} & \quad \text{Rd} \\
\hline
\hline
01
\end{array}
\]

FMINV <V><d>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;

Single-precision and double-precision

\[
\begin{array}{ccccccccccccccccccccccccccccccccccc}
\hline
0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & 1 & sz & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & \quad \text{Rn} & \quad \text{Rd} \\
\hline
\hline
01
\end{array}
\]

FMINV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

\(<V>\) For the half-precision variant: is the destination width specifier, H.
For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

\[
\begin{array}{ccccccccccccccccccccccccccccccccccc}
sz & <V> \\
\hline
0 & 5 & \text{RESERVED} \\
1 & \\
\end{array}
\]

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

\[
\begin{array}{ccccccccccccccccccccccccccccccccccc}
Q & <T> \\
\hline
0 & 4H \\
1 & 8H \\
\end{array}
\]
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

V[d] = Reduce(ReduceOp_FMIN, operand, esize);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision
(Armv8.2)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-------------------|
| 0 1 0 1 1 1 1 1 0 0 | L | M | Rm | 0 0 0 1 | H | 0 | Rn | Rd | 02 |
```

FMLA <Hd>, <Hn>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

Scalar, single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-------------------|
| 0 1 0 1 1 1 1 1 1 | sz | L | M | Rm | 0 0 0 1 | H | 0 | Rn | Rd | 02 |
```

FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
Vector, half-precision
(Armv8.2)

FMLA \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.H[index]}

\begin{verbatim}
if !\texttt{HaveFP16Ext()} then UNDEFINED;
integer idxdsize = if H == '1' then 128 else 64;
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
integer d = \texttt{UInt}(Rd);
integer index = \texttt{UInt}(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize \div esize;
boolean sub_op = (o2 == '1');
\end{verbatim}

Vector, single-precision and double-precision

FMLA \texttt{<Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[index]}

\begin{verbatim}
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = \texttt{UInt}(H:L);
    when '10' index = \texttt{UInt}(H);
    when '11' UNDEFINED;

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
ineger m = \texttt{UInt}(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 \ll \texttt{UInt}(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize \div esize;
boolean sub_op = (o2 == '1');
\end{verbatim}

Assembler Symbols

\begin{verbatim}
<Hz> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in “sz”:

\begin{array}{c|c}
sz & \texttt{V} \\
\hline
0 & S \\
1 & D \\
\end{array}

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":
\end{verbatim}
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsizesize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(Armv8.2)

```

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (a == '1');
```

Single-precision and double-precision

```

FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (op == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

```

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>
```

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

```
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<V_n>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<V_m>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);  
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLAL, FMLAL2 (by element)

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL (Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------|-------------------------------|-------------------------------|-------------------------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 0 | 0 | 0 | 0 | H | 0 | Rn | Rd |
| sz | S |

FMLAL \(<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]\)

if \(!\text{HaveFP16MulNoRoundingToFP32Ext}()\) then UNDEFINED;
integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}('0':Rm);\)  // Vm can only be in bottom 16 registers.
if \(sz == '1'\) then UNDEFINED;
integer index = \text{UInt}(H:L:M);

integer esize = 32;
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = \((S == '1')\);
integer part = 0;

FMLAL2 (Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------|-------------------------------|-------------------------------|-------------------------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | L | M | Rm | 1 | 0 | 0 | 0 | H | 0 | Rn | Rd |
| sz | S |

FMLAL2 \(<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]\)

if \(!\text{HaveFP16MulNoRoundingToFP32Ext}()\) then UNDEFINED;
integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}('0':Rm);\)  // Vm can only be in bottom 16 registers.
if \(sz == '1'\) then UNDEFINED;
integer index = \text{UInt}(H:L:M);

integer esize = 32;
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = \((S == '1')\);
integer part = 1;
Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

Operation

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements - 1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLAL, FMLAL2 (vector)

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | Rm  | 1  | 1  | 1  | 0  | 1  | 1  | Rn  | 1  | 1  | 0  | 0  | 1  | Rd |

S sz

FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;

FMLAL2
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | Rm  | 1  | 1  | 0  | 0  | 1  | 1  | Rn  | 1  | 1  | 0  | 0  | 1  | Rd |

S sz

FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```
FMLS (by element)

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision.

Scalar, half-precision

(Armv8.2)

```
<p>| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|-----------------|----------------|</p>
<table>
<thead>
<tr>
<th>0 1 0 1 1 1 1 1 1 0 0 L M</th>
<th>Rm 0 1 0 1 H 0</th>
<th>Rn Rd</th>
</tr>
</thead>
</table>
```

FMLS $<Hd>$, $<Hn>$, $<Vm>$.H[$<$index$>$]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');

Scalar, single-precision and double-precision

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|-----------------|----------------|
| 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 | sz L M | Rm 0 1 0 1 H 0 | Rn Rd |
|------------------------|-----------------|----------------|
```

FMLS $<V><d>$, $<V><n>$, $<V><m>.<Ts>$.[$<$index$>$]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
Vector, half-precision
(Armv8.2)

FMLS \langle \mathbf{V}_d \rangle., \langle \mathbf{T} \rangle., \langle \mathbf{V}_n \rangle., \langle \mathbf{T} \rangle., \langle \mathbf{V}_m \rangle..H[\langle \text{index} \rangle]

\begin{align*}
\text{if } &! \text{HaveFP16Ext()} \text{ then UNDEFINED; } \\
\text{integer } &\text{idxdsize } = \text{if } H == '1' \text{ then 128 else 64; } \\
\text{integer } &n = \text{UInt}(Rn); \\
\text{integer } &m = \text{UInt}(Rm); \\
\text{integer } &d = \text{UInt}(Rd); \\
\text{integer } &\text{index } = \text{UInt}(H:L:M); \\
\text{integer } &\text{esize } = 16; \\
\text{integer } &\text{datasize } = \text{if } Q == '1' \text{ then 128 else 64; } \\
\text{integer } &\text{elements } = \text{datasize DIV esize; } \\
\text{boolean } &\text{sub_op } = (o2 == '1');
\end{align*}

Vector, single-precision and double-precision

FMLS \langle \mathbf{V}_d \rangle., \langle \mathbf{T} \rangle., \langle \mathbf{V}_n \rangle., \langle \mathbf{T} \rangle., \langle \mathbf{V}_m \rangle..[\langle \text{index} \rangle]

\begin{align*}
\text{integer } &\text{idxdsize } = \text{if } H == '1' \text{ then 128 else 64; } \\
\text{integer } &\text{index } ; \\
\text{bit } &\text{Rmhi } = M; \\
\text{case } &sz:L \text{ of} \\
\text{when } &'0x' \text{ index } = \text{UInt}(H:L); \\
\text{when } &'10' \text{ index } = \text{UInt}(H); \\
\text{when } &'11' \text{ UNDEFINED; } \\
\text{integer } &d = \text{UInt}(Rd); \\
\text{integer } &n = \text{UInt}(Rn); \\
\text{integer } &m = \text{UInt}(Rmhi:Rm); \\
\text{if } &sz:Q == '10' \text{ then UNDEFINED; } \\
\text{integer } &\text{esize } = 32 << \text{UInt}(sz); \\
\text{integer } &\text{datasize } = \text{if } Q == '1' \text{ then 128 else 64; } \\
\text{integer } &\text{elements } = \text{datasize DIV esize; } \\
\text{boolean } &\text{sub_op } = (o2 == '1');
\end{align*}

Assembler Symbols

- \langle \mathbf{H}_d \rangle \quad \text{Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
- \langle \mathbf{H}_n \rangle \quad \text{Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.}
- \langle \mathbf{V} \rangle \quad \text{Is a width specifier, encoded in "sz":}
  \begin{array}{c|c}
    \text{sz} & \langle \mathbf{V} \rangle \\
  \hline
    0 & S \\
    1 & D \\
  \end{array}
- \langle d \rangle \quad \text{Is the number of the SIMD&FP destination register, encoded in the "Rd" field.}
- \langle n \rangle \quad \text{Is the number of the first SIMD&FP source register, encoded in the "Rn" field.}
- \langle \mathbf{V}_d \rangle \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}
- \langle T \rangle \quad \text{For the half-precision variant: is an arrangement specifier, encoded in "Q":}
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

<table>
<thead>
<tr>
<th>szL</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsizesize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bitempty(element1);
bitempty(element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bitempty(128) result = if merge then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in $FPCR$, the exception results in either a flag being set in $FPSR$, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

**Half-precision (Armv8.2)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | Rm | 0  | 0  | 0  | 0  | 1  | 1  | Rn | 1  | 1  | 0  | 0  | 1  | 1  | Rd |

```plaintext
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
esize = 16;
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV esize;
sub_op = (a == '1');
```

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 0  | 1  | 1  | Rn | 1  | 1  | 0  | 0  | 1  | 1  | Rd |

```plaintext
FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
esize = 32 << UInt(sz);
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV esize;
sub_op = (op == '1');
```

**Assembler Symbols**

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the half-precision variant: is an arrangement specifier, encoded in "Q":
  - | Q | `<T>` |
  -|----|------|
  -| 0  | 4H   |
  -| 1  | 8H   |
  
  For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

---

[FMLS (vector) Page 856](#)
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped. In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID AA64ISAR0_EL1. FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 1 1 1 1 1 1 0 1 0 1 0 0 H 0 Rn Rd

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);  // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;

FMLSL2
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 1 1 1 1 1 1 0 1 1 0 0 H 0 Rn Rd

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);  // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
Assembler Symbols

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<index>  Is the element index, encoded in the "H:L:M" fields.

<Ta>  Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<Tb>  Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize DIV 2];
  if sub_op then element1 = FPNeg(element1);
  Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLSL, FMLSL2 (vector)

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_AA64ISAR0_EL1. FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | Rn | 1 | 1 | 0 | 1 | 1 | Rd |

S sz

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;

FMLSL2
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | Rn | 1 | 1 | 0 | 0 | 1 | Rd |

S sz

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:
<Ta>
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2H</td>
</tr>
<tr>
<td>1</td>
<td>4H</td>
</tr>
</tbody>
</table>

Is an arrangement specifier, encoded in “Q”:

<Vn>
Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part];
bits(datasize DIV 2) operand2 = Vpart[m, part];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
for e = 0 to elements-1
  element1 = Elem[operand1, e, esize DIV 2];
  element2 = Elem[operand2, e, esize DIV 2];
  if sub_op then element1 = FPNeg(element1);
  Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (vector, immediate)

Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.
Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \textit{Half-precision} and \textit{Single-precision and double-precision}

\textbf{Half-precision}
\textit{(Armv8.2)}

\begin{verbatim}
                                  0 |  Q |  0  |  1  |  1  |  1  |  1  |  0  |  0  |  0  |  0  |  0  | Rd

FMOV <Vd>.<T>, #<imm>
if !HaveFP16Ext() then UNDEFINED;
integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);
imm = Replicate(imm16, datasize DIV 16);
\end{verbatim}

\textbf{Single-precision and double-precision}

\begin{verbatim}
                                  0 |  Q | op |  0  |  1  |  1  |  1  |  0  |  0  |  0  |  0  | Rd

Single-precision (op == 0)
FMOV <Vd>.<T>, #<imm>
Double-precision (Q == 1 && op == 1)
FMOV <Vd>.2D, #<imm>
\end{verbatim}

Assembler Symbols

\begin{verbatim}
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q";
\end{verbatim}
For the single-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in “a:b:c:d:e:f:g:h”. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions.

Operation

```c
CheckFPAdvSIMDEnabled64();

V[rd] = imm;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09, xml, sve v2020-09 rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | Rn  | Rd  | opc |

### Half-precision (ftype == 11) (Armv8.2)

FMOV <Hd>, <Hn>

### Single-precision (ftype == 00)

FMOV <Sd>, <Sn>

### Double-precision (ftype == 01)

FMOV <Dd>, <Dn>

```plaintext
ingredient d = UInt(Rd);
ingredient n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11' if HaveFP16Ext() then
    esize = 16;
  else
    UNDEFINED;
```

### Assembler Symbols

- `<Dd>` is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```plaintext
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[n];
Elem[Zeros() , 0 , esize] = operand;
V[d] = Zeros();
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | x  | 1  | 1  | x  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |
| rmode | opcode |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```
Half-precision to 32-bit (sf == 0 && ftype == 11 && rmode == 00 && opcode == 110)
(Armv8.2)

FMOV <Wd>, <Hn>

Half-precision to 64-bit (sf == 1 && ftype == 11 && rmode == 00 && opcode == 110)
(Armv8.2)

FMOV <Xd>, <Hn>

32-bit to half-precision (sf == 0 && ftype == 11 && rmode == 00 && opcode == 111)
(Armv8.2)

FMOV <Hd>, <Wn>

32-bit to single-precision (sf == 0 && ftype == 00 && rmode == 00 && opcode == 111)

FMOV <Sd>, <Wn>

Single-precision to 32-bit (sf == 0 && ftype == 00 && rmode == 00 && opcode == 110)

FMOV <Wd>, <Sn>

64-bit to half-precision (sf == 1 && ftype == 11 && rmode == 00 && opcode == 111)
(Armv8.2)

FMOV <Hd>, <Xn>

64-bit to double-precision (sf == 1 && ftype == 01 && rmode == 00 && opcode == 111)

FMOV <Dd>, <Xn>

64-bit to top half of 128-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 111)

FMOV <Vd>.D[1], <Xn>

Double-precision to 64-bit (sf == 1 && ftype == 01 && rmode == 00 && opcode == 110)

FMOV <Xd>, <Dn>

Top half of 128-bit to 64-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 110)

FMOV <Xd>, <Vn>.D[1]
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UNDEFINED;
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

case opcode<2:1>:rmode of
    when '00 xx'    // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'    // [US]CVTF
        rounding = FPRoundingMode(FPCR[]);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'    // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'    // FMOV
        if fltsize != 16 && fltsize != intsize then UNDEFINED;
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'    // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UNDEFINED;
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
        fltsize = 64;    // size of D[1] is 64
    when '11 11'    // FJCVTZS
        if !HaveFJCVTZSExt() then UNDEFINED;
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

FPCType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if op == \text{FPConvOp CVT ItoF} \&\& merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
  when \text{FPConvOp CVT FtoI}
    fltval = V[n];
    intval = \text{FPToFixed}(fltval, 0, unsigned, fpcr, rounding);
    X[d] = intval;
  when \text{FPConvOp CVT ItoF}
    intval = X[n];
    fltval = if merge then V[d] else Zeros();
    \text{Elem}[fltval, 0, fsize] = \text{FixedToFP}(intval, 0, unsigned, fpcr, rounding);
    V[d] = fltval;
  when \text{FPConvOp MOV FtoI}
    fltval = Vpart[n, part];
    intval = \text{ZeroExtend}(fltval, intsize);
    X[d] = intval;
  when \text{FPConvOp MOV ItoF}
    intval = X[n];
    fltval = intval<fsize-1:0>;
    Vpart[d, part] = fltval;
  when \text{FPConvOp CVT FtoI JS}
    bit Z;
    fltval = V[n];
    (intval, Z) = \text{FPToFixedJS}(fltval, fpcr, TRUE);
    PSTATE.<N,Z,C,V> = '0':Z:'00';
    X[d] = intval;
FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Assembly Format]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | imm8 | 1  | 0  | 0  | 0  | 0  | 0  | Rd |

Half-precision (ftype == 11) (Armv8.2)

FMOV <Hd>, #<imm>

Single-precision (ftype == 00)

FMOV <Sd>, #<imm>

Double-precision (ftype == 01)

FMOV <Dd>, #<imm>

integer d = UInt(Rd);

integer datasize;

```plaintext
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
    bits(datasize) imm = VFPExpandImm(imm8);
```

Assembler Symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<HD> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<SD> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see Modified immediate constants in A64 floating-point instructions.

Operation

CheckFPAdvSIMDEnabled64();

V[d] = imm;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09.rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ftype</td>
<td>0</td>
<td>Rm</td>
<td>1</td>
<td>Ra</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Half-precision (ftype == 11)
(ARMv8.2)

FMSUB <Hd>, <Hn>, <Hm>, <Ha>

Single-precision (ftype == 00)

FMSUB <Sd>, <Sn>, <Sm>, <Sa>

Double-precision (ftype == 01)

FMSUB <Dd>, <Dn>, <Dm>, <Da>

integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da> Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(esize) operand = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTYPE fpcr = FPCR[ ];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMUL (by element)

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in \textit{FPSCR}, the exception results in either a flag being set in \textit{FPSR} or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR_EL1}, \textit{CPTR_EL2}, and \textit{CPTR_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: \textit{Scalar, half-precision}, \textit{Scalar, single-precision and double-precision}, \textit{Vector, half-precision} and \textit{Vector, single-precision and double-precision}

### Scalar, half-precision

\textbf{(Armv8.2)}

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 1 1 1 1 0 0 | L | M | Rm | 1 0 0 1 | H | 0 | Rn | Rd |

\textbf{FMUL }<Hd>, <Hn>, <Vm>.H[<index>]

\textbf{if !HaveFP16Ext() then UNDEFINED;}

\textbf{integer idxdsize = if H == '1' then 128 else 64;}

\textbf{integer n = UInt(Rn);}

\textbf{integer m = UInt(Rm);}

\textbf{integer d = UInt(Rd);}

\textbf{integer index = UInt(H:L:M);}

\textbf{integer esize = 16;}

\textbf{integer datasize = esize;}

\textbf{integer elements = 1;}

\textbf{boolean mulx_op = (U == '1');}

### Scalar, single-precision and double-precision

\textbf{(Armv8.2)}

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 1 1 1 1 1 1 | sz | L | M | Rm | 1 0 0 1 | H | 0 | Rn | Rd |

\textbf{FMUL }<V>d, <V>n, <Vm>.<Ts>[<index>]

\textbf{integer idxdsize = if H == '1' then 128 else 64;}

\textbf{integer index;}

\textbf{bit Rmhi = M;}

\textbf{case sz:L of}

\textbf{when '0x' index = UInt(H:L);}

\textbf{when '10' index = UInt(H);}

\textbf{when '11' UNDEFINED;}

\textbf{integer d = UInt(Rd);}

\textbf{integer n = UInt(Rn);}

\textbf{integer m = UInt(Rmhi:Rm);}

\textbf{integer esize = 32 << UInt(sz);}

\textbf{integer datasize = esize;}

\textbf{integer elements = 1;}

\textbf{boolean mulx_op = (U == '1');}
**Vector, half-precision**

(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

U

FMUL \(<V_d>.<T>, <V_n>.<T>, <V_m>.H[<index>]\)

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

**Vector, single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | sz | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

U

FMUL \(<V_d>.<T>, <V_n>.<T>, <V_m>.<Ts>[<index>]\)

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
  when '0x' index = UInt(H:L);
  when '10' index = UInt(H);
  when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');

**Assembler Symbols**

| <Hd> | Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Hn> | Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| <V>  | Is a width specifier, encoded in "sz": |
|      | \( \begin{array}{c|c}
|      | sz \hline
|      | 0 | S \hline
|      | 1 | D \hline
|      | \end{array} \) |
| <d>  | Is the number of the SIMD&FP destination register, encoded in the "Rd" field. |
| <n>  | Is the number of the first SIMD&FP source register, encoded in the "Rn" field. |
| <Vd> | Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <T>  | For the half-precision variant: is an arrangement specifier, encoded in "Q": |
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Q Is an element size specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in "sz:L:H":

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(esize) element1;
bits(esize) element2 = Elem(operand2, index, esize);
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();
for e = 0 to elements-1
    element1 = Elem(operand1, e, esize);
    if multx op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);
V[d] = result;
```

**FMUL (vector)**

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 0 | 1 | 1 | 1 | Rn | Rd |

FMUL `<Vd>..<T>, <Vn>..<T>, <Vm>..<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

### Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1 | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 1 | 1 | 1 | Rn | Rd |

FMUL `<Vd>..<T>, <Vn>..<T>, <Vm>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

### Assembler Symbols

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 2S</td>
</tr>
<tr>
<td>0</td>
<td>1 4S</td>
</tr>
<tr>
<td>1</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

- `<Vn>` is the name of the first SIMD&FP source register, encoded in the "Rn" field.
Operation

\texttt{CheckFPAdvSIMDEnabled64()};

\begin{verbatim}
bits(datasize) operand1 = \texttt{V}[n];
bits(datasize) operand2 = \texttt{V}[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = \texttt{Elem}[operand1, e, esize];
    element2 = \texttt{Elem}[operand2, e, esize];
    \texttt{Elem}[result, e, esize] = \texttt{FPMul}(element1, element2, FPCR[ ]);
\end{verbatim}

\texttt{V}[d] = result;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | Rm | 0  | 0  | 0  | 0  | 1  | 0  | Rn | Rd |

Half-precision (ftype == 11)
(Armv8.2)

FMUL <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FMUL <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FMUL <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

bits(esize) product = FPMul(operand1, operand2, fpcr);
Elem[result, 0, esize] = product;
V[d] = result;
FMULX (by element)

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision, Scalar, single-precision and double-precision, Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U

FMULX <Hd>, <Hn>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsz = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');

Scalar, single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U

FMULX <V><d>, <V><n>, <Vm><Ts>[<index>]

integer idxdsz = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
   when '0x' index = UInt(H:L);
   when '10' index = UInt(H);
   when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
Vector, half-precision
(Armv8.2)

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 1 1 1 1 0 0 L M Rm 1 0 0 1 H 0 Rn Rd U
```

FMULX \(<V_d>\).<\(T>\), \(<V_n>\).<\(T>\), \(<V_m>\).H[<\(index>]\]

if \(!\text{HaveFP16Ext}()\) then UNDEFINED;

integer idxdsize = if \(H == '1'\) then 128 else 64;
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
integer \(d = \text{UInt}(Rd)\);
integer \(index = \text{UInt}(H:L:M)\);

integer esize = 16;
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;
boolean \(\text{mulx_op = (U == '1')}\);

Vector, single-precision and double-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 1 1 1 1 1 1 sz L M Rm 1 0 0 1 H 0 Rn Rd U
```

FMULX \(<V_d>\).<\(T>\), \(<V_n>\).<\(T>\), \(<V_m>\).<\(T_s>\)[<\(index>]\]

integer idxdsize = if \(H == '1'\) then 128 else 64;
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rmhi:Rm)\);
integer \(d = \text{UInt}(Rd)\);
integer \(sz = \text{UInt}(sz)\);
integer \(index;\)
bit \(Rmhi = M\);
case \(sz:L\) of
  when '0x' \(\text{index = UInt}(H:L)\);
  when '10' \(\text{index = UInt}(H)\);
  when '11' UNDEFINED;

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rmhi:Rm)\);
if \(sz:Q == '10'\) then UNDEFINED;
integer \(esize = 32 \ll \text{UInt}(sz)\);
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;
boolean \(\text{mulx_op = (U == '1')}\);

Assembler Symbols

- \(<H_d>\) Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<H_n>\) Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
- \(<V>\) Is a width specifier, encoded in “sz”:
  
<table>
<thead>
<tr>
<th>(sz)</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- \(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) For the half-precision variant: is an arrangement specifier, encoded in “Q”:
  
FMULX (by element)
For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

<table>
<thead>
<tr>
<th>Q</th>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the “H:L:M” fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

<table>
<thead>
<tr>
<th>sz</th>
<th>L</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>H:L</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMULX**

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

---

**Scalar half precision**
(ARMv8.2)

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 0 0 1 0 Rm 0 0 0 1 1 1 Rn Rd
```

FMULX <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

---

**Scalar single-precision and double-precision**

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 0 0 sz 1 Rm 1 1 0 1 1 1 Rn Rd
```

FMULX <V<d>, <V<n>, <V<m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

---

**Vector half precision**
(ARMv8.2)

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 1 0 0 1 0 Rm 0 0 0 1 1 1 Rn Rd
```

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMULX <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1

  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FMULX(element1, element2, fpcr);
V[d] = result;
FNEG (vector)

Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: 

**Half-precision**

\texttt{(Armv8.2)}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | Rd |
|    | U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\texttt{FNEG <Vd>.<T>, <Vn>.<T>}

if \(!\texttt{HaveFP16Ext}()\) then UNDEFINED;

integer \(d = \texttt{UInt}(\texttt{Rd});\)
integer \(n = \texttt{UInt}(\texttt{Rn});\)

integer esize = 16;
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = \((U == '1')\);

**Single-precision and double-precision**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | Rd |
|    | sz |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\texttt{FNEG <Vd>.<T>, <Vn>.<T>}

integer \(d = \texttt{UInt}(\texttt{Rd});\)
integer \(n = \texttt{UInt}(\texttt{Rn});\)

if \(sz:Q == '10'\) then UNDEFINED;
integer esize = 32 << \texttt{UInt}(sz);
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = \((U == '1')\);

**Assembler Symbols**

\texttt{<Vd>}  
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<T>}  
For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\texttt{<Vn>}  
Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 1 1 1 1 0 ftype 1 0 0 0 0 1 0 1 0 0 0 0 | Rn | Rd
opc
\end{verbatim}

Half-precision (ftype == 11) (Armv8.2)

\begin{verbatim}
FNEG <Hd>, <Hn>
\end{verbatim}

Single-precision (ftype == 00)

\begin{verbatim}
FNEG <Sd>, <Sn>
\end{verbatim}

Double-precision (ftype == 01)

\begin{verbatim}
FNEG <Dd>, <Dn>
\end{verbatim}

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
\end{verbatim}

Assembler Symbols

\begin{verbatim}
<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\end{verbatim}

Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

bits(esize) operand = V[n];
Elem[result, 0, esize] = FPNeg(operand);
V[d] = result;
\end{verbatim}
**FNMADD**

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Instruction Format](image)

**Half-precision (ftype == 11) (Armv8.2)**

FNMADD <Hd>, <Hn>, <Hm>, <Ha>

**Single-precision (ftype == 00)**

FNMADD <Sd>, <Sn>, <Sm>, <Sa>

**Double-precision (ftype == 01)**

FNMADD <Dd>, <Dn>, <Dm>, <Da>

```plaintext
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
e isize = 16;
    else
      UNDEFINED;
  end
```

**Assembler Symbols**

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Dn>** Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- **<Dm>** Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- **<Da>** Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
- **<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- **<Hm>** Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- **<Ha>** Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Sa> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(esize) operanda = V[a];
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bias(128) result = if merge then V[a] else Zeros();

operanda = FPNeg(operanda);
operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Dm>` Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
- `<Da>` Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
- `<Hm>` Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();

bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypetype fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a] else Zeros();

operand = FPNeg(operand);
Elem[result, 0, esize] = FPMulAdd(operand, operand1, operand2, fpcr);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | Rm  | 1  | 0  | 0  | 0  | 1  | 0  | Rn  | Rd  |
```

Half-precision (ftype == 11) (Armv8.2)

FNMUL <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FNMUL <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FNMUL <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
when '00' esize = 32;
when '01' esize = 64;
when '10' UNDEFINED;
when '11'
    if HaveFP16Ext() then
        esize = 16;
    else
        UNDEFINED;

Assembler Symbols

<Dd> is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRTypedefpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

bits(esize) product = FPMul(operand1, operand2, fpcr);
product = FPNeg(product);
Elem[result, 0, esize] = product;
V[d] = result;
```
FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

```
0 1 0 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 0

FRECPE <Hd>, <Hn>
```

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

```
0 1 0 1 1 1 1 0 1 sz 1 0 0 0 0 1 1 1 0 1 1 0

FRECPE <V><d>, <V><n>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision
(Armv8.2)

```
0 Q 0 0 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 0

FRECPE <Vd>.<T>, <Vn>.<T>
```

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR[]);
V[d] = result;
FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision
(Armv8.2)

FRECPS <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;

Scalar single-precision and double-precision

FRECPS <V>d>, <V>n>, <V>m

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Vector half precision
(Armv8.2)

FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

FRECPS  <Vd>.<T>,  <Vn>.<T>,  <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

define bits(datasize) operand1 = V[n];
define bits(datasize) operand2 = V[m];
define bits(esize) element1;
define bits(esize) element2;
define FPCRTypedef fpcr = FPCR[];
define boolean merge = elements == 1 && IsMerging(fpcr);
define bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
  define element1 = Elem[operand1, e, esize];
  define element2 = Elem[operand2, e, esize];
  define Elem[result, e, esize] = FPRecipStepFused(element1, element2);

\[ V[d] = \text{result}; \]
FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FRECPX <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | sz | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |

FRECPX <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
<br> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \( V[n] \);

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then \( V[d] \) else Zeros();
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecpX(element, fpcr);

\( V[d] = result; \)
FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register. A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(Armv8.5)

FRINT32X <Vd>,<T>, <Vn>.

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
FRINT32X (scalar)

Floating-point Round to 32-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rd |

ftype op

Rn

Single-precision (ftype == 00)

FRINT32X <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT32X <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Db> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Db> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Db> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMD Enabled64();

FPCRTypedefpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 32);

V[d] = result;
**FRINT32Z (vector)**

Floating-point Round to 32-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Vector single-precision and double-precision**

(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | sz| 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |

FRINT32Z <Vd>,<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[ ]);;

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
   element = Elem[operand, e, esize];
   Elem[result, e, esize] = FRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
FRINT32Z (scalar)

Floating-point Round to 32-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(Armv8.5)

Single-precision (ftype == 00)

FRINT32Z <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT32Z <Dd>, <Dn>

Assembler Symbols

<Db> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Db> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Db> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Db> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);

bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 32);
V[d] = result;
FRINT64X (vector)

Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(Armv8.5)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0   | Q | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | Rn | Rd |
| U   | op |

FRINT64X <Vd>.<T>, <Vn>.<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;
FRINT64X (scalar)

Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register. A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |

- **ftype** - Type of instruction
- **op** - Operation code
- **Rn** - 64-bit name of the SIMD&FP source register
- **Rd** - 64-bit name of the SIMD&FP destination register

Single-precision (ftype == 00)

```plaintext
FRINT64X <Sd>, <Sn>
```

Double-precision (ftype == 01)

```plaintext
FRINT64X <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

- `<Dd>` - Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` - Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` - Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` - Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypedefpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 64);

V[d] = result;
FRINT64Z (vector)

Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(Armv8.5)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U | 0  | Q  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | Rn | Rd |
| op|     |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FRINT64Z <Vd>..<T>, <Vn>..<T>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;
for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
FRINT64Z (scalar)

Floating-point Round to 64-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(Armv8.5)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |

ftype op

Single-precision (ftype == 00)

FRINT64Z <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT64Z <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;

Assembler Symbols

<DD> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<DN> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<SDD> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<SNN> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bite(128) result = if merge then V[d] else Zeros();
bite(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 64);
V[d] = result;
FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

**Armv8.2**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FRINTA <Vd>,<T>, <Vn>,<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

### Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDeno1o2n; 
  when '100' rounding = FPRounding_TIEAWAY; 
  when '101' UNDEFINED; 
  when '110' rounding = FPRoundingMode(FPCR()); exact = TRUE; 
  when '111' rounding = FPRoundingMode(FPCR());
end case;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

\[
\begin{array}{c|c}
\text{Q} & \text{T} \\
0 & 4H \\
1 & 8H \\
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\[
\begin{array}{c|c|c}
\text{sz} & \text{Q} & \text{T} \\
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & \text{RESERVED} \\
1 & 1 & 2D \\
\end{array}
\]

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d] = result;
FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in 

FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | ftype | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | Rn | 0  | 0  | 0  | 0  | 0  | Rd |

rmode

Half-precision (ftype == 11)
(Armv8.2)

FRINTA <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTA <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTA <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypemotion = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then \texttt{V[d]} else \texttt{Zeros}();
bits(esize) operand = \texttt{V[n]};

\texttt{Elem}[, result, 0, esize] = \texttt{FPRoundInt}(operand, fpcr, \texttt{FPRounding\_TIEAWAY}, \texttt{FALSE});
\texttt{V[d]} = result;
**FRINTI (vector)**

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the **FPCR**, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Half-precision** and **Single-precision and double-precision**

---

**Half-precision** *(Armv8.2)*

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |

**FRINTI <Vd>.<T>, <Vn>.<T>**

if !**HaveFP16Ext**() then UNDEFINED;

integer d = **UInt**(Rd);
integer n = **UInt**(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;

**FPRounding** rounding;

**case U:o1:o2** of

when '0xx' rounding = **FPDecodeRounding**(o1:o2);
when '100' rounding = **FPRounding_TIEAWAY**;
when '101' UNDEFINED;
when '110' rounding = **FPRoundingMode**(FPCR[0]); exact = TRUE;
when '111' rounding = **FPRoundingMode**(FPCR[0]);

---

**Single-precision and double-precision**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d] = result;
FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Half-precision (ftype == 11)

(Armv8.2)

FRINTI <Hd>, <Hn>

### Single-precision (ftype == 00)

FRINTI <Sd>, <Sn>

### Double-precision (ftype == 01)

FRINTI <Dd>, <Dn>

```plaintext
type d = UInt(Rd);
n = UInt(Rn);

integer esize;

case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols:

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypedefs fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bites(128) result = if merge then V[d] else Zeros();
bites(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd
U o2 o1

FRINTM <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Single-precision and double-precision

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | sz 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd
U o2 o1
integer \(d = \text{UInt}(\text{Rd})\);
integer \(n = \text{UInt}(\text{Rn})\);

if \(sz:Q == '10'\) then UNDEFINED;
integer esize = 32 \ll\ \text{UInt}(sz);
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = \text{FALSE};
\text{FPRounding} rounding;
case U:o1:o2 of

when '0xx' rounding = \text{FPDecodeRounding}(o1:o2);
when '100' rounding = \text{FPRounding_TIEAWAY};
when '101' UNDEFINED;
when '110' rounding = \text{FPRoundingMode}(\text{FPCR}[]); exact = \text{TRUE};
when '111' rounding = \text{FPRoundingMode}(\text{FPCR}[]);

\textbf{Assembler Symbols}

\begin{tabular}{|c|c|}
\hline
\textbf{Q} & \textbf{<T>} \\
\hline
0 & 4H \\
1 & 8H \\
\hline
\end{tabular}

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\begin{tabular}{|c|c|c|}
\hline
\textbf{sz} & \textbf{Q} & \textbf{<T>} \\
\hline
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & RESERVED \\
1 & 1 & 2D \\
\hline
\end{tabular}

\textbf{Operation}

\textbf{CheckFPAdvSIMDEnabled64}();
bits(datasize) operand = \textbf{V}[[n]];
bits(datasize) result;
bits(esize) element;

for \(e = 0\) to elements-1

\begin{verbatim}
    element = \textbf{Elem}[operand,e,esize];
    \textbf{Elem}[result,e,esize] = \textbf{FPRoundInt}(element,\textbf{FPCR}[],rounding,exact);
\end{verbatim}

\textbf{V}[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | 0 | 1 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | Rn | Rd | rmode

Half-precision (ftype == 11) (Armv8.2)

FRINTM <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTM <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTM <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
when '00' esize = 32;
when '01' esize = 64;
when '10' UNDEFINED;
when '11'
  if HaveFP16Ext() then
    esize = 16;
  else
    UNDEFINED;
FPRounding rounding;
rounding = FPDoseedRounding('10');

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypet fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bis(128) result = if merge then V[d] else Zeros();
bis(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision

(ARMv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|
| O | O | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd |
| U | 0 | 2 | o1 |

FRINTN <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;

FPRounding rounding;

### Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------|---------------------|
| O | O | 0 | 1 | 1 | 1 | 0 | 0 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | Rn | Rd |
| U | 0 | 2 | o1 |
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Assembling Symbols

<Pd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;
FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | 0 0 1 1 1 1 0 | ftype | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Rn | Rd |
| rmode |

**Half-precision (ftype == 11)**

(Armv8.2)

FRINTN <Hd>, <Hn>

**Single-precision (ftype == 00)**

FRINTN <Sd>, <Sn>

**Double-precision (ftype == 01)**

FRINTN <Dd>, <Dn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;

FP Rounding rounding;
rounding = FPDetectRounding('00');
```

Assembler Symbols

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypedef pc = FPCR[];
boolean merge = IsMerging(fpc);
bits(128) result = if merge then $V[d]$ else Zeros();
bits(esize) operand = $V[n]$;

Elem[result, 0, esize] = FPRoundInt(operand, fpc, rounding, FALSE);

$V[d]$ = result;
FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | | Rn | Rd |
| U  | o2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FRINTP <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;

FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[0]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[0]);

Single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |
| U  | o2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;
FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |   ftype | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Rd |
```

Half-precision (ftype == 11)  
(Armv8.2)

FRINTP <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTP <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTP <Dd>, <Dn>

```plaintext
type d = UInt(Rd);
type n = UInt(Rn);

type esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

FProunding rounding;
rounding = FPDecodeRounding('01');
```

Assembler Symbols

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Dn>** Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Sn>** Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypemfpcr = FPCR[];
booleandomerge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(pagesize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d] = result;
FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the \textit{FPCR}, and writes the result to the SIMD&FP destination register.

When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in \textit{FPCR}, the exception results in either a flag being set in \textit{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \textit{CPACR EL1}, \textit{CPTR EL2}, and \textit{CPTR EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \textit{Half-precision} and \textit{Single-precision and double-precision}

\textbf{Half-precision (Armv8.2)}

\begin{verbatim}
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |
| U | 02 | o1 |

FRINTX <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:01:02 of
when '0xx' rounding = FPDecodeRounding(o1:o2);
when '100' rounding = FPRounding_TIEAWAY;
when '101' UNDEFINED;
when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
when '111' rounding = FPRoundingMode(FPCR[]);
\end{verbatim}

\textbf{Single-precision and double-precision}

\begin{verbatim}
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
| U | 02 | o1 |
\end{verbatim}
FRINTX \(<Vd>.<T>, \ <Vn>.<T>\)

integer \(d = \text{UInt}(\text{Rd})\);
integer \(n = \text{UInt}(\text{Rn})\);

if sz:Q == '10' then UNDEFINED;
integer \(e\text{size} = 32 \ll \text{UInt}(\text{sz})\);
integer \(\text{datasize} = \text{if Q == '1' then 128 else 64}\);
integer \(\text{elements} = \text{datasize} \div \text{esize}\);

boolean \(\text{exact} = \text{FALSE};\)
strain Rounding \(\text{rounding}\);
case U:o1:o2 of
    when '0xx' \(\text{rounding} = \text{FPDecodeRounding}(\text{o1:o2})\);
    when '100' \(\text{rounding} = \text{FPRounding_TIEAWAY}\);
    when '101' \text{UNDEFINED};
    when '110' \(\text{rounding} = \text{FPRoundingMode}(\text{FPCR[]}); \text{exact} = \text{TRUE};\)
    when '111' \(\text{rounding} = \text{FPRoundingMode}(\text{FPCR[]});\)

\textbf{Assembler Symbols}

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\textbf{Operation}

\texttt{\textbf{CheckFPAdvSIMDEnabled64}();}

\texttt{bits(datasize) operand = V[n];}
\texttt{bits(datasize) result;}
\texttt{bits(esize) element;}

\texttt{for e = 0 to elements-1}
\quad \texttt{element =} \texttt{Elem[operand, e, esize];}
\quad \texttt{Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);}
\texttt{V[d] = result;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the FPCR, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 0 0 0 1 1 1 1 0 ftype 1 0 0 1 1 1 0 1 0 0 0 0 Rn | Rd |
| rmode |

**Half-precision (ftype == 11)**

(Armv8.2)

FRINTX <Hd>, <Hn>

**Single-precision (ftype == 00)**

FRINTX <Sd>, <Sn>

**Double-precision (ftype == 01)**

FRINTX <Dd>, <Dn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);
```

**Assembler Symbols**

- `<Dd>` Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Dn>` Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Hd>` Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Hn>` Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Sd>` Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Sn>` Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

\textbf{CheckFPAdvSIMEnabled64}();

\textbf{FPCRTyp}e \ fpcr = \text{FPCR}[];
boolean \ merge = \text{IsMerging}(\text{fpcr});
bites(128) \ result = \text{if} \ merge \ \text{then} \ V[d] \ \text{else} \ Zeros();
bites(\text{esize}) \ operand = V[n];

\textbf{Elem}[\text{result}, 0, \text{esize}] = \text{FPRoundInt}(\text{operand}, \text{fpcr}, \text{rounding}, \text{TRUE});

\text{V[d]} = \text{result};
FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

### Half-precision
(ARMv8.2)

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  |
| U | o2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

### Single-precision and double-precision

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  |
| U | o2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
  when '0xx' rounding = FPDecodeRounding(o1:o2);
  when '100' rounding = FPRounding_TIEAWAY;
  when '101' UNDEFINED;
  when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
  when '111' rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 | 0 | 0 | 1 | 1 | 1 | 0 | ftype | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

| Rd | Rn |

Half-precision (ftype == 11)
(Armv8.2)

FRINTZ <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTZ <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTZ <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;

\[
\begin{align*}
\text{case } ftype \text{ of} \\
\quad \text{when '00' esize = 32;} \\
\quad \text{when '01' esize = 64;} \\
\quad \text{when '10' undefined;} \\
\quad \text{when '11'} \\
\quad \quad \text{if HaveFP16Ext() then} \\
\quad \quad \quad \text{esize = 16;} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{undefined;}
\end{align*}
\]

FPRounding rounding;

rounding = FPDDecodeRounding('11');

Assembler Symbols

<\text{Dd}> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<\text{Dn}> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\text{Hd}> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<\text{Hn}> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<\text{Sd}> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<\text{Sn}> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypelfcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();
bits(esize) operand = V[n];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);
V[d] = result;
Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see [Floating-point exception traps](#).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: [Scalar half precision](#), [Scalar single-precision and double-precision](#), [Vector half precision](#) and [Vector single-precision and double-precision](#).

### Scalar half precision

(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 0 1 0 0 1 1 0 1 0</td>
</tr>
</tbody>
</table>

|Rn| Rd
---|---

FRSQRTE <Hd>, <Hn>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

### Scalar single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0 1 0 0 0 0 1 1 1 0 1 1 0 1 0 0 1 1 0 1 0 0</td>
</tr>
</tbody>
</table>

|Rn| Rd
---|---

FRSQRTE <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

### Vector half precision

(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

|Rn| Rd
---|---

FRSQRTE <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
Vector single-precision and double-precision

FRSQRT \langle Vd \rangle, \langle T \rangle, \langle Vn \rangle, \langle T \rangle

integer d = \text{ UInt}(Rd);
integer n = \text{ UInt}(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 \ll \text{ UInt}(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize \text{ DIV} esize;

Assembler Symbols

\langle Hd \rangle Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
\langle Hn \rangle Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
\langle V \rangle Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>\langle V \rangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

\langle d \rangle Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
\langle n \rangle Is the number of the SIMD&FP source register, encoded in the "Rn" field.
\langle Vd \rangle Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\langle T \rangle For the half-precision variant: is an arrangement specifier, encoded in "Q":

\langle T \rangle

<table>
<thead>
<tr>
<th>Q</th>
<th>\langle T \rangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>\langle T \rangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\langle Vn \rangle Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \langle V \rangle[n];

bits(esize) element;
FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then \langle V \rangle[d] else \langle Zeros \rangle();

for e = 0 to elements-1
  element = \langle Elem \rangle[operand, e, esize];
  \langle Elem \rangle[result, e, esize] = FPRsqrTEstimate(element, fpcr);

\langle V \rangle[d] = result;
FRRSQTS

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

Scalar half precision

(Armv8.2)

|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| 0 1 0 1 1 1 1 0 1 1 0                       | Rm 0 0 1 1 1 1                               |
|                                             | Rd |

FRRSQTS <Hd>, <Hn>, <Hm>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
exteger n = UInt(Rn);
exteger m = UInt(Rm);
exteger esize = 16;
exteger datasize = esize;
exteger elements = 1;

Scalar single-precision and double-precision

(Armv8.2)

|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------------------------|-----------------------------------------------|
| 0 1 0 1 1 1 1 0 1 1 1 0 1 sz 1                             | Rm 1 1 1 1 1 1                               |
|                                                               | Rd |

FRRSQTS <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
exteger n = UInt(Rn);
exteger m = UInt(Rm);
exteger esize = 32 << UInt(sz);
exteger datasize = esize;
exteger elements = 1;

Vector half precision

(Armv8.2)

|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------------------------|-----------------------------------------------|
| 0 Q 0 0 1 1 1 0 1 1 0                                       | Rm 0 0 1 1 1 1                               |
|                                                               | Rd |

FRRSQTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
exteger n = UInt(Rn);
exteger m = UInt(Rm);
exteger esize = 16;
exteger datasize = if Q == '1' then 128 else 64;
exteger elements = datasize DIV esize;
## Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

FRSQRTS \(<Vd>, <T>, <Vn>, <T>, <Vm>, <T>\)

```plaintext
generate C code:
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

### Assembler Symbols

| \(<Hd>\) | Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field. |
| \(<Hn>\) | Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field. |
| \(<Hm>\) | Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field. |
| \(<V>\)  | Is a width specifier, encoded in "sz": |
| sz      | <V> |
| 0       | 0 S |
| 1       | 1 D |

| \(<d>\) | Is the number of the SIMD&FP destination register, in the "Rd" field. |
| \(<n>\) | Is the number of the first SIMD&FP source register, encoded in the "Rn" field. |
| \(<m>\) | Is the number of the second SIMD&FP source register, encoded in the "Rm" field. |
| \(<Vd>\) | Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |
| \(<T>\)  | For the half-precision variant: is an arrangement specifier, encoded in "Q": |
| Q        | <T> |
| 0        | 4H |
| 1        | 8H |

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

| \(<Vn>\) | Is the name of the first SIMD&FP source register, encoded in the "Rn" field. |
| \(<Vm>\) | Is the name of the second SIMD&FP source register, encoded in the "Rm" field. |
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esimal) element1;
bits(esimal) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);
V[d] = result;
FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision
(Armv8.2)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rd |

FSQRT <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Single-precision and double-precision

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rd |

FSQRT <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>2S</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>4S</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>2D</td>
<td></td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

`CheckFPAdvSIMDEnabled64();`

```c
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR[]);
```

`V[d] = result;`
**FSQRT (scalar)**

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```plaintext
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ftype</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Half-precision (ftype == 11)**

(Armv8.2)

FSQRT <Hd>, <Hn>

**Single-precision (ftype == 00)**

FSQRT <Sd>, <Sn>

**Double-precision (ftype == 01)**

FSQRT <Dd>, <Dn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;

case ftype of
  when '00' esize = 32;
  when '01' esize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      esize = 16;
    else
      UNDEFINED;
endcase
```

**Assembler Symbols**

- **<Dd>** Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Dn>** Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Hd>** Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Hn>** Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
- **<Sd>** Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Sn>** Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTYPE fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

bits(ewise) operand = V[n];

Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d] = result;
FSUB (vector)

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register. This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision.

**Half-precision**

(Armv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>O</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

**Single-precision and double-precision**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>O</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>sz</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

FSUB (vector)
<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRType fpcr = FPCR[];
bits(datasize) result;

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  diff = FPSub(element1, element2, fpcr);
  Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>1</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

Half-precision (ftype == 11)
(Armv8.2)

FSUB <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FSUB <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FSUB <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(esize) operand1 = V[n];
bits(esize) operand2 = V[m];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n] else Zeros();

Elem(result, 0, esize) = FPSub(operand1, operand2, fpcr);
V[d] = result;
INS (element)

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias `MOV (element)`.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 1 1 0 0 0 0 | imm5 | 0 | imm4 | 1 | Rd | Rn
```

`INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]`

- `integer d = UInt(Rd);`
- `integer n = UInt(Rn);`
- `integer size = LowestSetBit(imm5);`
- `if size > 3 then UNDEFINED;`
- `integer dst_index = UInt(imm5<4:size+1>);`
- `integer src_index = UInt(imm4<3:size>);`
- `integer idxdxsize = if imm4<3> == '1' then 128 else 64;`
- `// imm4<size-1:0> is IGNORED`
- `integer esize = 8 << size;`

Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<Ts>** Is an element size specifier, encoded in “imm5”:
  
<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<index1>** Is the destination element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- **<index2>** Is the source element index encoded in “imm5:imm4”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index2&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm4&lt;3:0&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm4&lt;3:1&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm4&lt;3:2&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm4&lt;3&gt;</td>
</tr>
</tbody>
</table>

Unspecified bits in “imm4” are ignored but should be set to zero by an assembler.
Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]
\[
\text{bits(idxdsize) operand} = \text{V}[n];
\]
\[
\text{bits(128) result};
\]
\[
\text{result} = \text{V}[d];
\]
\[
\text{Elem}[\text{result, dst\_index, esize}] = \text{Elem}[\text{operand, src\_index, esize}];
\]
\[
\text{V}[d] = \text{result};
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias \texttt{MOV (from general)}.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | imm5| 0  | 0  | 0  | 1  | 1  | Rn |   |   |   | Rd |

\texttt{INS \langle Vd \rangle.<Ts>[\langle index \rangle], \langle R \rangle<\langle n \rangle}

integer \( d = \texttt{UInt}(\text{Rd}); \)
integer \( n = \texttt{UInt}(\text{Rn}); \)

integer \( \text{size} = \texttt{LowestSetBit}(\text{imm5}); \)

if \( \text{size} > 3 \) then UNDEFINED;
integer \( \text{index} = \texttt{UInt}(\text{imm5}<4:\text{size}+1>); \)

integer \( \text{esize} = 8 \ll \text{size}; \)

\textbf{Assembler Symbols}

\texttt{<Vd>} \quad Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<Ts>} \quad Is an element size specifier, encoded in “imm5”:

\begin{center}
\begin{tabular}{c|c}
\hline
\texttt{imm5} & \texttt{<Ts>} \\
\hline
x0000 & RESERVED \\
xxx1 & B \\
xxx10 & H \\
x100 & S \\
x1000 & D \\
\hline
\end{tabular}
\end{center}

\texttt{<index>} \quad Is the element index encoded in “imm5”:

\begin{center}
\begin{tabular}{c|c}
\hline
\texttt{imm5} & \texttt{<index>} \\
\hline
x0000 & RESERVED \\
xxx1 & \texttt{imm5}<4:1> \\
xxx10 & \texttt{imm5}<4:2> \\
x100 & \texttt{imm5}<4:3> \\
x1000 & \texttt{imm5}<4> \\
\hline
\end{tabular}
\end{center}

\texttt{<R>} \quad Is the width specifier for the general-purpose source register, encoded in “imm5”:

\begin{center}
\begin{tabular}{c|c}
\hline
\texttt{imm5} & \texttt{<R>} \\
\hline
x0000 & RESERVED \\
xxx1 & W \\
xxx10 & W \\
x100 & W \\
x1000 & X \\
\hline
\end{tabular}
\end{center}

\texttt{<n>} \quad Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.
Operation

\(\text{CheckFPAdvSIMDEnabled64}()\);
bits(esize) element = \(X[n]\);
bits(128) result;

result = \(V[d]\);
\(\text{Elem}[\text{result, index, esize}] = \text{element}\);
\(V[d] = \text{result}\);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: \texttt{No offset} and \texttt{Post-index}

**No offset**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | x  | size | Rn  |    |    |    |    |    |    |    |    |    |    |

L  \hspace{1cm} \text{opcode}

**One register (opcode == 0111)**

LD1 \{ <Vt>.<T> \}, \[<Xn|SP>\]

**Two registers (opcode == 1010)**

LD1 \{ <Vt>.<T>, <Vt2>.<T> \}, \[<Xn|SP>\]

**Three registers (opcode == 0110)**

LD1 \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> \}, \[<Xn|SP>\]

**Four registers (opcode == 0010)**

LD1 \{ <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> \}, \[<Xn|SP>\]

integer t = \texttt{UInt}(Rt);
integer n = \texttt{UInt}(Rn);
integer m = integer \texttt{UNKNOWN};
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

**Post-index**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | Rm  | x  | x  | 1  | x  | size | Rn  |    |    |    |    |    |    |    |    |    |    |

L  \hspace{1cm} \text{opcode}
One register, immediate offset (Rm == 11111 & opcode == 0111)
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>

One register, register offset (Rm != 11111 & opcode == 0111)
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>

Two registers, immediate offset (Rm == 11111 & opcode == 1010)
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Two registers, register offset (Rm != 11111 & opcode == 1010)
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

Three registers, immediate offset (Rm == 11111 & opcode == 0110)
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

Three registers, register offset (Rm != 11111 & opcode == 0110)
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

Four registers, immediate offset (Rm == 11111 & opcode == 0010)
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Four registers, register offset (Rm != 11111 & opcode == 0010)
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in "Q":

LD1 (multiple structures)
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#8</td>
</tr>
<tr>
<td>1</td>
<td>#16</td>
</tr>
</tbody>
</table>

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

LD1 (multiple structures)
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
                tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

8-bit (opcode == 000)

LD1 {<Vt>.B}[<index>], [<Xn|SP>]

16-bit (opcode == 010 && size == x0)

LD1 {<Vt>.H}[<index>], [<Xn|SP>]

32-bit (opcode == 100 && size == 00)

LD1 {<Vt>.S}[<index>], [<Xn|SP>]

64-bit (opcode == 100 && S == 0 && size == 01)

LD1 {<Vt>.D}[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
8-bit, immediate offset (Rm == 11111 && opcode == 000)
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1

8-bit, register offset (Rm != 11111 && opcode == 000)
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
    scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register. Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

**No offset**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | size | Rn | Rt |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| L | R | opcode | S |

LD1R { <Vt>,<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

**Post-index**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | size | Rm | 1 | 1 | 0 | 0 | size | Rn | Rt |
|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| L | R | opcode | S |

**Immediate offset (Rm == 11111)**

LD1R { <Vt>,<T> }, [<Xn|SP>], <imm>

**Register offset (Rm != 11111)**

LD1R { <Vt>,<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

**Assembler Symbols**

- `<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` Is an arrangement specifier, encoded in “size:Q”:

```
  size | Q | <T>
  00   0 | 8B
  00   1 | 16B
  01   0 | 4H
  01   1 | 8H
  10   0 | 2S
  10   1 | 4S
  11   0 | 1D
  11   1 | 2D
```

- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the post-index immediate offset, encoded in “size”:
size | <imm>
--- | ---
00 | #1
01 | #2
10 | #4
11 | #8

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

**No offset**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

LD2 { <Vt>.,<T> }, [<Xn|SP>]

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Immediate offset (Rm == 1111)

LD2 { <Vt>.,<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 1111)

LD2 { <Vt>.,<T> }, [<Xn|SP>], <Xm>

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

**Assembler Symbols**

**<Vt>**

Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

**<T>**

Is an arrangement specifier, encoded in "size:Q"

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

**<Vt2>**

Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

**<Xn|SP>**

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;  // number of iterations
integer selem;  // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;  // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;  // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;  // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;  // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1;  // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;  // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;  // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

LD2 (multiple structures)
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LD2 (single structure)**

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

**No offset**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | O  | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |
```

L R opcode

**8-bit (opcode == 000)**

LD2 `{ <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP]>`

**16-bit (opcode == 010 && size == x0)**

LD2 `{ <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP]>`

**32-bit (opcode == 100 && size == 00)**

LD2 `{ <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP]>`

**64-bit (opcode == 100 && S == 0 && size == 01)**

LD2 `{ <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP]>`


```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

**Post-index**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | O  | 0 | 1 | 1 | 0 | 1 | 1 | 1 | Rm | x | x | 0 | S | size | Rn | Rt |
```

L R opcode
8-bit, immediate offset (Rm == 11111 && opcode == 000)
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2

8-bit, register offset (Rm != 11111 && opcode == 000)
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == x0)
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8

32-bit, register offset (Rm != 11111 && opcode == 100 && size == x0)
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
:index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);   // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);   // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);      // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);      // D[0-1]
    scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(128) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

### No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | size | 8B |
| L | R | opcode | S |

LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

### Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | size | 8B |
| L | R | opcode | S |

Immediate offset (Rm == 11111)

LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

### Assembler Symbols

- **<Vt>** Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- **<Vt2>** Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<imm>** Is the post-index immediate offset, encoded in “size”:
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

### Shared Decode

```plaintext
definitions:
    integer init_scale = UInt(opcode<2:1>);
    integer scale = init_scale;
    integer selem = UInt(opcode<0>:R) + 1;
    boolean replicate = FALSE;
    integer index;

    case scale of
        when 3
            // load and replicate
            if L == '0' || S == '1' then UNDEFINED;
            scale = UInt(size);
            replicate = TRUE;
        when 0
            index = UInt(Q:S:size);  // B[0-15]
        when 1
            if size<0> == '1' then UNDEFINED;
            index = UInt(Q:S:size<1>);  // H[0-7]
        when 2
            if size<1> == '1' then UNDEFINED;
            if size<0> == '0' then
                index = UInt(Q:S);  // S[0-3]
            else
                if S == '1' then UNDEFINED;
                index = UInt(Q);  // D[0-1]
            scale = 3;
        MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
        integer datasize = if Q == '1' then 128 else 64;
        integer esize = 8 << scale;
```
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
        t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:

A is a packed array of 3-element structures. Each element is a 16-bit halfword.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

L  opcode

LD3 {<Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

L  opcode

Immediate offset (Rm == 11111)

LD3 {<Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD3 {<Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
Assembler Symbols

<\textit{Vt}> Is the name of the first or only SIMD\&FP register to be transferred, encoded in the "Rt" field.

<\textit{T}> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>\textit{T}</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\textit{Vt}2> Is the name of the second SIMD\&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<\textit{Vt}3> Is the name of the third SIMD\&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<\textit{Xn}|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<\textit{imm}> Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>\textit{imm}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<\textit{Xm}> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

\begin{verbatim}
    MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
    integer datasize = if Q == '1' then 128 else 64;
    integer esize = 8 << UInt(size);
    integer elements = datasize DIV esize;

    integer rpt; // number of iterations
    integer selem; // structure elements

    case opcode of
    when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
    otherwise UNDEFINED;

    // .1D format only permitted with LD1 & ST1
    if size:Q == '110' && selem != 1 then UNDEFINED;
\end{verbatim}
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

offs = Zeros();
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
        offs = offs + ebytes;
    tt = (tt + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LD3 (single structure)**

Load single 3-element structure to one lane of three registers. This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn | Rt |

L  R  opcode (opcode == 001)


**16-bit (opcode == 011 && size == x0)**

LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]

**32-bit (opcode == 101 && size == 00)**


**64-bit (opcode == 101 && S == 0 && size == 01)**

LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn | Rt |

L  R  opcode

LD3 (single structure)  Page 987
8-bit, immediate offset (Rm == 11111 && opcode == 001)
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3

8-bit, register offset (Rm != 11111 && opcode == 001)
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(128) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index.

### No offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>L</td>
</tr>
</tbody>
</table>

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

- integer \( t = \text{UInt}(Rt); \)
- integer \( n = \text{UInt}(Rn); \)
- integer \( m = \text{integer UNKNOWN}; \)
- boolean \( wback = \text{FALSE}; \)
- boolean \( \text{tag\_checked} = wback \oplus n \neq 31; \)

### Post-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>L</td>
</tr>
</tbody>
</table>

### Immediate offset (Rm == 11111)

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

### Register offset (Rm != 11111)

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

- integer \( t = \text{UInt}(Rt); \)
- integer \( n = \text{UInt}(Rn); \)
- integer \( m = \text{UInt}(Rm); \)
- boolean \( wback = \text{TRUE}; \)
- boolean \( \text{tag\_checked} = wback \oplus n \neq 31; \)

### Assembler Symbols

- \(<Vt>\) Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- \(<T>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- \(<Vt2>\) Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- \(<Vt3>\) Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- \(<Xn|SP>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#3</td>
</tr>
<tr>
<td>01</td>
<td>#6</td>
</tr>
<tr>
<td>10</td>
<td>#12</td>
</tr>
<tr>
<td>11</td>
<td>#24</td>
</tr>
</tbody>
</table>

<imm> Is the post-index immediate offset, encoded in “size”:

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```plaintext
text
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
    scale = 3;

  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
  integer datasize = if Q == '1' then 128 else 64;
  integer esize = 8 << scale;
```
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LD4 (multiple structures)**

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see LD3 (multiple structures).

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

**No offset**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | size | Rn  | Rt  |


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

**Post-index**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | size | Rn  | Rt  |

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

**Immediate offset (Rm == 11111)**

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

**Register offset (Rm != 11111)**

LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

**Assembler Symbols**

- `<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- `<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << Uint(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
  when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

LD4 (multiple structures)
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

offs = Zeros();
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
      offs = offs + ebytes;
      tt = (tt + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| L  | R  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| opcode |

#### 8-bit (opcode == 001)


#### 16-bit (opcode == 011 && size == x0)


#### 32-bit (opcode == 101 && size == 00)


#### 64-bit (opcode == 101 && S == 0 && size == 01)


```java
integer t = UInt(Rt);
integer n = UInt(Rn);
n = integer UNKNOWN;
boolean wback = FALSE;
tag_checked = wback || n != 31;
```

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| L  | R  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| opcode |

8-bit, immediate offset (Rm == 11111 && opcode == 001)

8-bit, register offset (Rm != 11111 && opcode == 001)

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in “Q:S:size”.
For the 16-bit variant: is the element index, encoded in “Q:S:size<1>”.
For the 32-bit variant: is the element index, encoded in “Q:S”.
For the 64-bit variant: is the element index, encoded in “Q”.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;
case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
    scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
    // replicate to fill 128- or 64-bit register
    V[t] = Replicate(element, datasize DIV esize);
    offs = offs + ebytes;
    t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
    if memop == MemOp_LOAD then
        // insert into one lane of 128-bit register
        Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[t] = rval;
    else // memop == MemOp_STORE
        // extract from one lane of 128-bit register
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  |    |     |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| L | R  | opcode | S    |

LD4R { <Vt>,<T>, <Vt2>,<T>, <Vt3>,<T>, <Vt4>,<T> }, [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| L | R  | opcode | S    |

Immediate offset (Rm == 11111)

LD4R { <Vt>,<T>, <Vt2>,<T>, <Vt3>,<T>, <Vt4>,<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD4R { <Vt>,<T>, <Vt2>,<T>, <Vt3>,<T>, <Vt4>,<T> }, [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Is the post-index immediate offset, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#4</td>
</tr>
<tr>
<td>01</td>
<td>#8</td>
</tr>
<tr>
<td>10</td>
<td>#16</td>
</tr>
<tr>
<td>11</td>
<td>#32</td>
</tr>
</tbody>
</table>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
in
integer index;
case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);     // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);        // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);          // D[0-1]
      scale = 3;
memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
datasize = if Q == '1' then 128 else 64;
esize = 8 << scale;
```
if haveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(128) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPadding();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**LDNP (SIMD&FP)**

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Depending on the configuration, the instruction may have different behaviors.

### 32-bit (opc == 00)

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

### 64-bit (opc == 01)

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

### 128-bit (opc == 10)

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

// Empty.

For information about the CONstrained UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP (SIMD&FP)*.

**Assembler Symbols**

- `<Dt1>` is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Dt2>` is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<Qt1>` is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Qt2>` is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<St1>` is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<St2>` is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as `<imm>/4`.

For the 64-bit variant:

- For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as `<imm>/8`.

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as `<imm>/16`. 
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

Operation

CheckFPAdvSIMEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
else
    address = X[n];
address = address + offset;
data1 = Mem[address, dbytes, AccType_VECSTREAM];
data2 = Mem[address+dbytes, dbytes, AccType_VECSTREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

**Post-index**

<table>
<thead>
<tr>
<th>opc</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm7</td>
<td>Rt2</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

LDP <St1>, <St2>, [<Xn|SP>], #<imm>

64-bit (opc == 01)

LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

128-bit (opc == 10)

LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;

**Pre-index**

<table>
<thead>
<tr>
<th>opc</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm7</td>
<td>Rt2</td>
</tr>
</tbody>
</table>

32-bit (opc == 00)

LDP <St1>, <St2>, [<Xn|SP>], #<imm>!

64-bit (opc == 01)

LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>!

128-bit (opc == 10)

LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>!

boolean wback = TRUE;
boolean postindex = FALSE;

**Signed offset**

<table>
<thead>
<tr>
<th>opc</th>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm7</td>
<td>Rt2</td>
</tr>
</tbody>
</table>

LDP (SIMD&FP)
32-bit (opc == 00)
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]  

64-bit (opc == 01)
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]  

128-bit (opc == 10)
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]  

boolean wback = FALSE;
boolean postindex = FALSE;

For information about the constrained unpredictable behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDP (SIMD&FP).

Assembler Symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the “imm7” field as <imm>/4.
For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the “imm7” field as <imm>/4.
For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the “imm7” field as <imm>/8.
For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the “imm7” field as <imm>/8.
For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the “imm7” field as <imm>/16.
For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the “imm7” field as <imm>/16.
Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bv(64) offset = lsl(signextend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = constrainunpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE; // result is UNKNOWN
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP EndOfInstruction();

Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
data1 = Mem[address, dbytes, AccType_VEC];
data2 = Mem[address+dbytes, dbytes, AccType_VEC];
if rt_unknown then
data1 = bits(datasize) UNKNOWN;
data2 = bits(datasize) UNKNOWN;
V[t] = data1;
V[t2] = data2;
if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

Post-index

<table>
<thead>
<tr>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

| 31   | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1    | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

8-bit (size == 00 && opc == 01)

LDR <Bt>, [<Xn|SP>], #<simm>

16-bit (size == 01 && opc == 01)

LDR <Ht>, [<Xn|SP>], #<simm>

32-bit (size == 10 && opc == 01)

LDR <St>, [<Xn|SP>], #<simm>

64-bit (size == 11 && opc == 01)

LDR <Dt>, [<Xn|SP>], #<simm>

128-bit (size == 00 && opc == 11)

LDR <Qt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Pre-index

<table>
<thead>
<tr>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

opc
8-bit (size == 00 && opc == 01)
LDR <Bt>, [<Xn|SP>, #<simm>]

16-bit (size == 01 && opc == 01)
LDR <Ht>, [<Xn|SP>, #<simm>]

32-bit (size == 10 && opc == 01)
LDR <St>, [<Xn|SP>, #<simm>]

64-bit (size == 11 && opc == 01)
LDR <Dt>, [<Xn|SP>, #<simm>]

128-bit (size == 00 && opc == 11)
LDR <Qt>, [<Xn|SP>, #<simm>]

boolean wback = TRUE;
boolean postindex = FALSE;
ingernt scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bite(64) offset = SignExtend(imm9, 64);

Unsigned offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|
| size 1 1 1 1 0 1 x 1 imm12 imm12 | Rn 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 |
| opc 1 1 1 1 0 1 imm12 imm12      | Rt 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 |

8-bit (size == 00 && opc == 01)
LDR <Bt>, [<Xn|SP>{, #<pimm}>]

16-bit (size == 01 && opc == 01)
LDR <Ht>, [<Xn|SP>{, #<pimm}>]

32-bit (size == 10 && opc == 01)
LDR <St>, [<Xn|SP>{, #<pimm}>]

64-bit (size == 11 && opc == 01)
LDR <Dt>, [<Xn|SP>{, #<pimm}>]

128-bit (size == 00 && opc == 11)
LDR <Qt>, [<Xn|SP>{, #<pimm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
ingernt scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bite(64) offset = LSL(ZeroExtend(imm12, 64), scale);
Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<It> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| opc| 0  | 1  | 1  | 1  | 0  | 0  | imm19|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

32-bit (opc == 00)

LDR <St>, <label>

64-bit (opc == 01)

LDR <Dt>, <label>

128-bit (opc == 10)

LDR <Qt>, <label>

integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
  when '00'
    size = 4;
  when '01'
    size = 8;
  when '10'
    size = 16;
  when '11'
    UNDEFINED;
offset = SignExtend(imm19:'00', 64);

Assembler Symbols

< Dt > Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
< Qt > Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
< St > Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
< label > Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

bits(64) address = PC[] + offset;
bits(size*8) data;
if HaveMTEExt() then
  SetTagCheckedInstruction(FALSE);
CheckFPAdvSIMDEnabled64();
data = Mem[address, size, AccType_VEC];
V[t] = data;
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

- `<Bt>` Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.
- `<Dt>` Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.
- `<Ht>` Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.
- `<Qt>` Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.
- `<St>` Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the “Rt” field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the “Rn” field.
- `<Wm>` When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the “Rm” field.
- `<Xm>` When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the “Rm” field.
- `<extend>` For the 8-bit variant: is the index extend specifier, encoded in “option”: integer scale = UInt(opc<1>:size);
  if scale > 4 then UNDEFINED;
  if option<1> == '0' then UNDEFINED; // sub-word index
  ExtendType extend_type = DecodeRegExtend(option);
  integer shift = if S == '1' then scale else 0;
<table>
<thead>
<tr>
<th>Option</th>
<th>Extend</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

<table>
<thead>
<tr>
<th>Option</th>
<th>Extend</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>011</td>
<td>LSL</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

<amount>

For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#4</td>
</tr>
</tbody>
</table>

**Shared Decode**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tagChecked = memop != MemOp_PREFETCH;
```
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>imm9</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 01)

LDUR <Bt>, [<Xn|SP>{, #<simm>}

16-bit (size == 01 && opc == 01)

LDUR <Ht>, [<Xn|SP>{, #<simm>}

32-bit (size == 10 && opc == 01)

LDUR <St>, [<Xn|SP>{, #<simm>}

64-bit (size == 11 && opc == 01)

LDUR <Dt>, [<Xn|SP>{, #<simm>}

128-bit (size == 00 && opc == 11)

LDUR <Qt>, [<Xn|SP>{, #<simm>}

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Bt>  Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>  Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>  Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

address = address + offset;

case memop of
  when MemOp_STORE
    data = V[t];
    Mem[address, datasize DIV 8, AccType_VEC] = data;
  when MemOp_LOAD
    data = Mem[address, datasize DIV 8, AccType_VEC];
    V[t] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<T>} Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\texttt{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\texttt{<Vm>} Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

\texttt{<Ts>} Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\texttt{<index>} Is the element index, encoded in "size:L:H:M":
### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0   Q   1   0   1   1   1   size   L   M   Rm   0   1   0   0   H   0   Rn   Rd   o2
```

MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```plaintext
integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

```
size   Q   <T>
00     x   RESERVED
01     0   4H
01     1   8H
10     0   2S
10     1   4S
11     x   RESERVED
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

```
size   <Vm>
00     RESERVED
01     0:Rm
10     M:Rm
11     RESERVED
```

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

```
size   <Ts>
00     RESERVED
01     H
10     S
11     RESERVED
```

<index> Is the element index, encoded in "size:L:H:M":

```
### Operation

CheckFPAdvSIMDEnabled64();

```c
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
|   | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Q | | | | | | | | | | | | | | |
| 0 | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 1 | 0 | 1 | Rd |
```

MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');

### Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (scalar)

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.

Depending on the settings in the \textit{CPACR\_EL1}, \textit{CPTR\_EL2}, and \textit{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of \textit{DUP (element)}. This means:

- The encodings in this description are named to match the encodings of \textit{DUP (element)}.
- The description of \textit{DUP (element)} gives the operational pseudocode for this instruction.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 0 0 0 0 | imm5 | 0 0 0 0 0 1 | Rn | Rd
\end{verbatim}

\texttt{MOV <V><d>, <Vn>.<T>[<index>]}

is equivalent to

\texttt{DUP <V><d>, <Vn>.<T>[<index>]}

and is always the preferred disassembly.

\textbf{Assembler Symbols}

- \texttt{<V>} Is the destination width specifier, encoded in “imm5”:
  \begin{verbatim}
  imm5  <V>
  \end{verbatim}
  \begin{itemize}
  \item x0000: RESERVED
  \item xxx1: B
  \item xxx10: H
  \item xx100: S
  \item x1000: D
  \end{itemize}

- \texttt{<d>} Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- \texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- \texttt{<T>} Is the element width specifier, encoded in “imm5”:
  \begin{verbatim}
  imm5  <T>
  \end{verbatim}
  \begin{itemize}
  \item x0000: RESERVED
  \item xxx1: B
  \item xxx10: H
  \item xx100: S
  \item x1000: D
  \end{itemize}

- \texttt{<index>} Is the element index encoded in “imm5”:
  \begin{verbatim}
  imm5  <index>
  \end{verbatim}
  \begin{itemize}
  \item x0000: RESERVED
  \item xxx1: imm5<4:1>
  \item xxx10: imm5<4:2>
  \item xx100: imm5<4:3>
  \item x1000: imm5<4>
  \end{itemize}

\textbf{Operation}

The description of \textit{DUP (element)} gives the operational pseudocode for this instruction.

\textbf{Operational information}

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  \begin{itemize}
  \item The values of the data supplied in any of its registers.
  \item The values of the NZCV flags.
  \end{itemize}
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register. This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of `INS (element)`. This means:

- The encodings in this description are named to match the encodings of `INS (element)`.
- The description of `INS (element)` gives the operational pseudocode for this instruction.

```
|0|1|1|0|1|1|0|0|0|imm5|0|imm4|1|Rn|Rd|
```

MOV `<Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]`

is equivalent to

`INS `<Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]`

and is always the preferred disassembly.

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ts>` Is an element size specifier, encoded in "imm5":
  - `imm5 <Ts>`
    - x0000: RESERVED
    - xxx1: B
    - xxx10: H
    - xx100: S
    - x1000: D
- `<index1>` Is the destination element index encoded in “imm5”:  
  - `imm5 <index1>`
    - x0000: RESERVED
    - xxx1: imm5<4:1>
    - xxx10: imm5<4:2>
    - xx100: imm5<4:3>
    - x1000: imm5<4>
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.
- `<index2>` Is the source element index encoded in “imm5:imm4”:
  - `imm5 <index2>`
    - x0000: RESERVED
    - xxx1: imm4<3:0>
    - xxx10: imm4<3:1>
    - xx100: imm4<3:2>
    - x1000: imm4<3>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

The description of `INS (element)` gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.
This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.
Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of `INS (general)`. This means:

- The encodings in this description are named to match the encodings of `INS (general)`.
- The description of `INS (general)` gives the operational pseudocode for this instruction.

```plaintext
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

MOV <Vd>.<Ts>[<index>], <R><n>

is equivalent to

`INS <Vd>.<Ts>[<index>], <R><n>`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ts>` Is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<index>` Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>x1000</td>
<td>imm5&lt;4&gt;</td>
</tr>
</tbody>
</table>

- `<R>` Is the width specifier for the general-purpose source register, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>W</td>
</tr>
<tr>
<td>xxx10</td>
<td>W</td>
</tr>
<tr>
<td>xx100</td>
<td>W</td>
</tr>
<tr>
<td>x1000</td>
<td>X</td>
</tr>
</tbody>
</table>

- `<n>` Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

**Operation**

The description of `INS (general)` gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of ORR (vector, register). This means:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

MOV <Vd>.<T>, <Vn>.<T>

is equivalent to

ORR <Vd>.<T>, <Vn>.<T>, <Vn>.<T>

and is the preferred disassembly when Rm == Rn.

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of UMOV. This means:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode for this instruction.

32-bit (Q == 0 && imm5 == xx100)

MOV <Wd>, <Vn>.S[<index>]

is equivalent to

UMOV <Wd>, <Vn>.S[<index>]

and is always the preferred disassembly.

64-reg,UMOV-64-reg (Q == 1 && imm5 == x1000)

MOV <Xd>, <Vn>.D[<index>]

is equivalent to

UMOV <Xd>, <Vn>.D[<index>]

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> For the 32-bit variant: is the element index encoded in "imm5<4:3>".

For the 64-reg,UMOV-64-reg variant: is the element index encoded in "imm5<4>".

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>Q</th>
<th>Op</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>cmode</th>
<th>0</th>
<th>1</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>Rd</th>
</tr>
</thead>
</table>

8-bit (op == 0 && cmode == 1110)

`MOVI <Vd>,<T>, #<imm8>{, LSL #0}`

16-bit shifted immediate (op == 0 && cmode == 100)

`MOVI <Vd>,<T>, #<imm8>{, LSL #<amount>}`

32-bit shifted immediate (op == 0 && cmode == 000)

`MOVI <Vd>,<T>, #<imm8>{, LSL #<amount>}`

32-bit shifting ones (op == 0 && cmode == 110)

`MOVI <Vd>,<T>, #<imm8>, MSL #<amount>`

64-bit scalar (Q == 0 && op == 1 && cmode == 1110)

`MOVI <Dd>, #<imm>`

64-bit vector (Q == 1 && op == 1 && cmode == 1110)

`MOVI <Vd>,2D, #<imm>`

```plaintext
integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0000' operation = ImmediateOp_MOVI;
  when '0001' operation = ImmediateOp_MVNI;
  when '0010' operation = ImmediateOp_ORR;
  when '0011' operation = ImmediateOp_BIC;
  when '0100' operation = ImmediateOp_MOVI;
  when '0101' operation = ImmediateOp_MVNI;
  when '0110' operation = ImmediateOp_ORR;
  when '0111' operation = ImmediateOp_BIC;
  when '1000' operation = ImmediateOp_MOVI;
  when '1001' operation = ImmediateOp_MVNI;
  when '1010' operation = ImmediateOp_ORR;
  when '1011' operation = ImmediateOp_BIC;
  when '1100' operation = ImmediateOp_MOVI;
  when '1101' operation = ImmediateOp_MVNI;
  when '1110' operation = ImmediateOp_MOVI;
  when '1111' operation = ImmediateOp_MOVI;

// FMOV Dn,#imm is in main FP instruction set
if Q == '0' then UNDEFINED;
  operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```
Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<imm> Is a 64-bit immediate 'aaaaaaaaabbbbbbbbbccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by
the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the
destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<index> Is the element index, encoded in "size:L:H:M":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

   element2 = UInt(Elem[operand2, index, esize]);
   for e = 0 to elements-1
      element1 = UInt(Elem[operand1, e, esize]);
      product = (element1*element2)<esize-1:0>;
      Elem[result, e, esize] = product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

- \( \text{MUL} <Vd>.,<T>, <Vn>.,<T>, <Vm>.,<T> \)

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean poly = (U == '1');
```

**Assembler Symbols**

- \(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \(<T>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- \(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MVN**

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of **NOT**. This means:

- The encodings in this description are named to match the encodings of **NOT**.
- The description of **NOT** gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |

**MVN** `<Vd>.<T>`, `<Vn>.<T>`

is equivalent to

**NOT** `<Vd>.<T>`, `<Vn>.<T>`

and is always the preferred disassembly.

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

The description of **NOT** gives the operational pseudocode for this instruction.

**Operational information**

If `PSTATE.DIT` is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the $\text{CPACR\_EL1}$, $\text{CPTR\_EL2}$, and $\text{CPTR\_EL3}$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 16-bit shifted immediate (cmode == 10x0)

$$\text{MVNI} \ <Vd>.<T>, \ #\langle\text{imm8}\rangle\{, \ \text{LSL} \ #\langle\text{amount}\rangle\}$$

### 32-bit shifted immediate (cmode == 0xx0)

$$\text{MVNI} \ <Vd>.<T>, \ #\langle\text{imm8}\rangle\{, \ \text{LSL} \ #\langle\text{amount}\rangle\}$$

### 32-bit shifting ones (cmode == 110x)

$$\text{MVNI} \ <Vd>.<T>, \ #\langle\text{imm8}\rangle, \ \text{MSL} \ #\langle\text{amount}\rangle$$

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;  
bits(datasize) imm;  
bits(64) imm64;

```plaintext
ImmediateOp operation;
```

case cmode:op of
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11111'  // FMOV Dn,#imm is in main FP instruction set  
    if Q == '0' then UNDEFINED;  
    operation = ImmediateOp_MOVI;

imm64 = AdvSIMDEExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);

### Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** For the 16-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

- **<imm8>** Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".
- **<amount>** For the 16-bit shifted immediate variant: is the shift amount encoded in "cmode<1>":

`
defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

<table>
<thead>
<tr>
<th>cmode&lt;0&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>0 1 0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

NEG <V<d>, <V<n>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>0 1 0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

NEG <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q".
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
for e = 0 to elements-1
  element = SInt(Elem[operand, e, esize]);
  if neg then
    element = -element;
  else
    element = Abs(element);
  Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**NOT**

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias `MVN`.

```
NOT <Vd>,<T>, <Vn>,<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = NOT(element);

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>Rm</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "Q":
  
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
operand2 = NOT(operand2);
result = operand1 OR operand2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ORR (vector, immediate)**

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### 16-bit (cmode == 10x1)

ORR `<Vd>.<T>`, `#<imm8>{, LSL #<amount>}`

### 32-bit (cmode == 0xx1)

ORR `<Vd>.<T>`, `#<imm8>{, LSL #<amount>}`

integer `rd` = `UInt`(Rd);

integer `datasize` = if Q == '1' then 128 else 64;

bits(`datasize`) `imm`;

bits(64) `imm64`;

`ImmediateOp` operation;

case cmode:op of
when '0xx00' operation = `ImmediateOp_MOVI`;
when '0xx10' operation = `ImmediateOp_ORR`;
when '10x00' operation = `ImmediateOp_MOVI`;
when '10x10' operation = `ImmediateOp_ORR`;
when '110x0' operation = `ImmediateOp_MOVI`;
when '1110x' operation = `ImmediateOp_MOVI`;
when '11110' operation = `ImmediateOp_MOVI`;

`imm64` = `AdvSIMDExpandImm`(op, cmode, a:b:c:d:e:f:g:h);

`imm` = `Replicate`(imm64, `datasize DIV 64`);

---

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP register, encoded in the "Rd" field.
- `<T>` For the 16-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</td>
</tr>
</tbody>
</table>

- `<imm8>` Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".
- `<amount>` For the 16-bit variant: is the shift amount encoded in "cmode<1>":

<table>
<thead>
<tr>
<th><code>cmode&lt;1&gt;</code></th>
<th><code>&lt;amount&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in "cmode<2:1>":
<table>
<thead>
<tr>
<th>cmode&lt;2:1&gt;</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>24</td>
</tr>
</tbody>
</table>

defaulting to 0 if LSL is omitted.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
  when ImmediateOp_MOVI
    result = imm;
  when ImmediateOp_MVNI
    result = NOT(imm);
  when ImmediateOp_ORR
    operand = V[rd];
    result = operand OR imm;
  when ImmediateOp_BIC
    operand = V[rd];
    result = operand AND NOT(imm);

V[rd] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 0 | 1 | 1 | 0 | 1 | 0 | 1 | Rm | 0 | 0 | 0 | 1 | 1 | 1 | Rn | Rd | size |

ORR <Vd>..<T>, <Vn>..<T>, <Vm>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;

Assembler Symbols

Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector)</td>
<td>Rm == Rn</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = operand1 OR operand2;

V[d] = result;

Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over \{0, 1\}*. Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean poly = (U == '1');

### Assembly Symbols

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```assembly
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**PMULL, PMULL2**

Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see *Polynomial arithmetic over \{0, 1\}*. 

The PMULL instruction extracts each source vector from the lower half of each source register, while the PMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1, CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 1 | 1 | 0 | 0 | 0 | Rn | Rd |

PMULL(2) <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !HaveBit128PMULLExt() then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

### Assembler Symbols

| 2 | Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:
| Q | 2 |
| 0 | [absent] |
| 1 | [present] |

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
  element1 = Elem[operand1, e, esize];
  element2 = Elem[operand2, e, esize];
  Elem[result, e, 2*esize] = PolynomialMult(element1, element2);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see ADDHN.

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

### Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RAX1

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SHA3 is implemented.

Advanced SIMD
(Armv8.2)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | Rm |
| 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | Rn |
| 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | Rd |

RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
V[d] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**RBIT (vector)**

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| Rd | Rn |

RBIT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
ninteger n = UInt(Rn);

integer esize = 8;
ninteger datasize = if Q == '1' then 128 else 64;
ninteger elements = datasize DIV 8;
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "Q":
  ```
  Q  <T>
  0  8B
  1  16B
  ```
- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```r
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bvets(datasize) result;
bvets(esize) element;
bvets(esize) rev;

for e = 0 to elements-1
  element = Evm[operand, e, esize];
  for i = 0 to esize-1
    rev<esize-1-i> = element<i>;
  Evm[result, e, esize] = rev;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 0 0 1 1 0 | size 1 0 0 0 0 0 0 0 1 1 0 | Rd
0 | Rn
```

REV16 $<Vd>$.<T>, $<Vn>$.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=enze: B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

**Assembler Symbols**

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
iconteger element = 0;
iconteger rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**REV32 (vector)**

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | Rn |   |   |   |   |   |   |   |   |
```

**REV32** `<Vd>`..<T>, `<Vn>`..<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=size: B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//    8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//    8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

**Assembler Symbols**

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>1x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0
size Rd Rn U
```

The `REV64 <Vd>.<T>, <Vn>.<T>` instruction is described as follows:

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=size: B(0), H(1), S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
// 64=B = 0, 64+H = 1, 64+S = 2, 64+D = X
// 32+B = 1, 32+H = 2, 32+S = X, 32+D = X
// 16+B = 2, 16+H = X, 16+S = X, 16+D = X
// 8+B = X, 8+H = X, 8+S = X, 8+D = X
// => 3-(op+size) (index bits in group)
// 64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
// 32+B = 2, 32+H = 1, 32+S = X, 32+D = X
// 16+B = 1, 16+H = X, 16+S = X, 16+D = X
// 8+B = X, 8+H = X, 8+S = X, 8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
  when '10' container_size = 16;
  when '01' container_size = 32;
  when '00' container_size = 64;
end;

integer containers = datasize DIV container_size;
icontainer.elements_per_container = container_size DIV esize;
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SHRN.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | != | 0000 | immh | 1  | 0  | 0  | 0  | 1  | 1  | Rn | Rd |   |   |   |   |   |   |   |   |   |   |

RSHRN(2) <Vd>, <Tb>, <Vn>, <Ta>, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

- 2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
Q  2
 0  [absent]
 1  [present]
```

- <Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.
- <Tb> Is an arrangement specifier, encoded in “immh:Q”:

```
<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified intermediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8H</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- <Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.
- <Ta> Is an arrangement specifier, encoded in “immh”:

```
<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified intermediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

- <shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”: 0
### Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see SUBHN.

The RSUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Example:

```
RSUBHN(2) <Vd>, <Tb>, <Ta>, <Vm>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

### Assembler Symbols

- **Q**
  - Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":
  ```plaintext
  \[
  \begin{array}{c|c}
  \text{Q} & 2 \\
  \hline
  0 & \text{[absent]} \\
  1 & \text{[present]} \\
  \end{array}
  \]
  
  - **<Vd>**
    - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
  
  - **<Tb>**
    - Is an arrangement specifier, encoded in "size:Q":
      ```plaintext
      \[
      \begin{array}{c|c|c}
      \text{size} & \text{Q} & <Tb> \\
      \hline
      00 & 0 & 8B \\
      00 & 1 & 16B \\
      01 & 0 & 4H \\
      01 & 1 & 8H \\
      10 & 0 & 2S \\
      10 & 1 & 4S \\
      11 & x & \text{RESERVED} \\
      \end{array}
      \]
      
      - **<Vn>**
        - Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
      
      - **<Ta>**
        - Is an arrangement specifier, encoded in "size":
          ```plaintext
          \[
          \begin{array}{c|c}
          \text{size} & <Ta> \\
          \hline
          00 & 8H \\
          01 & 4S \\
          10 & 2D \\
          11 & \text{RESERVED} \\
          \end{array}
          \]
          
      - **<Vm>**
        - Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
```

RSUBHN, RSUBHN2
Operation

\texttt{CheckFPAdvSIMDEnabled64()};
\texttt{bits(2*datasize) operand1 = \texttt{V[n]};}
\texttt{bits(2*datasize) operand2 = \texttt{V[m]};}
\texttt{bits(datasize) result;}
\texttt{integer round\_const = if round then 1 \ll (esize - 1) else 0;}
\texttt{bits(2*esize) element1;}
\texttt{bits(2*esize) element2;}
\texttt{bits(2*esize) sum;}

\texttt{for e = 0 to elements-1}
\texttt{element1 = \texttt{Elem[operand1, e, 2*esize]};}
\texttt{element2 = \texttt{Elem[operand2, e, 2*esize]};}
\texttt{if sub\_op then}
\texttt{sum = element1 - element2;}
\texttt{else}
\texttt{sum = element1 + element2;}
\texttt{sum = sum + round\_const;}
\texttt{\texttt{Elem[result, e, esize] = sum<2*esize-1:esize>;}}
\texttt{Vpart[d, part] = result;}

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

If PSTATE.DIT is 1:

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register, while the SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>Rm</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rn</th>
<th></th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SABAL{2} <Vd>,<Ta>, <Vn>,<Tb>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
Q  2
  0 [absent]
  1 [present]
```

<Vd>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  Is an arrangement specifier, encoded in "size":

```
size <Ta>
  00  8H
  01  4S
  10  2D
  11  RESERVED
```

<Vn>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>  Is an arrangement specifier, encoded in "size:Q":

```
size    Q <Tb>
  00    0  8B
  00    1  16B
  01    0  4H
  01    1  8H
  10    0  2S
  10    1  4S
  11    x  RESERVED
```

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

If PSTATE.DIT is 1:

\[ \text{SABD} \]

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 1 | 1 | 0 | 1 | Rn | Rd | ac |

SABD `<Vd>`. `<T>`, `<Vn>`.`<T>`, `<Vm>`.`<T>

integer `d` = UInt(Rd);
integer `n` = UInt(Rn);
integer `m` = UInt(Rm);
if size == '11' then UNDEFINED;
integer `esize` = 8 << UInt(size);
integer `datasize` = if Q == '1' then 128 else 64;
integer `elements` = datasize DIV esize;

boolean `unsigned` = (U == '1');
boolean `accumulate` = (ac == '1');

Assembler Symbols

`<Vd>`  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>`  Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

`<Vn>`  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

`<Vm>`  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\text{bits(datasize) operand1 = V[n];}
\]

\[
\text{bits(datasize) operand2 = V[m];}
\]

\[
\text{bits(datasize) result;}
\]

\[
\text{integer element1;}
\]

\[
\text{integer element2;}
\]

\[
\text{bits(esize) absdiff;}
\]

\[
\text{result = if accumulate then V[d] else Zeros();}
\]

\[
\text{for e = 0 to elements-1}
\]

\[
\text{element1 = Int(Elem[operand1, e, esize], unsigned);}
\]

\[
\text{element2 = Int(Elem[operand2, e, esize], unsigned);}
\]

\[
\text{absdiff = Abs(element1 - element2)[esize-1:0];}
\]

\[
\text{Elem[result, e, esize] = Elem[result, e, esize] + absdiff;}
\]

\[
V[d] = result;
\]

Operational information

If PSTATE.DIT is 1:
The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction writes the vector to the lower half of the destination register and clears the upper half, while the SABDL2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\textbf{CheckFPAdvSIMDEnabled64}();
b\text{bits(datamize) operand1} = \text{Vpart}[n, \text{part}];
b\text{bits(datamize) operand2} = \text{Vpart}[m, \text{part}];
b\text{bits(2*datamize) result};
integer element1;
integer element2;
b\text{bits(2*esize) absdiff};

\text{result} = \text{if accumulate then V}\{d\} \text{ else Zeros(); }
\text{for e = 0 to elements-1 }
\begin{align*}
\text{element1} &= \text{Int}(\text{Elem}[\text{operand1, e, esize}], \text{unsigned}); \\
\text{element2} &= \text{Int}(\text{Elem}[\text{operand2, e, esize}], \text{unsigned}); \\
\text{absdiff} &= \text{Abs}(\text{element1-element2})<2*\text{esize}-1:0>;
\text{Elem}[\text{result, e, 2*esize}] = \text{Elem}[\text{result, e, 2*esize}] + \text{absdiff};
\end{align*}
\text{V}\{d\} = \text{result};

Operational information

If PSTATE.DIT is 1:
  \begin{itemize}
    \item The execution time of this instruction is independent of:
      \begin{itemize}
        \item The values of the data supplied in any of its registers.
        \item The values of the NZCV flags.
      \end{itemize}
    \item The response of this instruction to asynchronous exceptions does not vary based on:
      \begin{itemize}
        \item The values of the data supplied in any of its registers.
        \item The values of the NZCV flags.
      \end{itemize}
  \end{itemize}
SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

| <Vd> | Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <Ta> | Is an arrangement specifier, encoded in "size:Q": |
| <Vn> | Is the name of the SIMD&FP source register, encoded in the "Rn" field. |
| <Tb> | Is an arrangement specifier, encoded in "size:Q": |

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

```
<AsmTable
| size | Q | <Ta>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

```
<AsmTable
| size | Q | <Tb>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```
Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\text{bits}(\text{datasize}) \text{ operand} = V[n];
\]

\[
\text{bits}(\text{datasize}) \text{ result};
\]

\[
\text{bits}(2*\text{esize}) \text{ sum};
\]

\[
\text{integer} \text{ op1};
\]

\[
\text{integer} \text{ op2};
\]

\[
\text{if acc then result} = V[d];
\]

\[
\text{for e = 0 to elements-1}
\]

\[
\text{op1} = \text{Int(Elem[operand, 2*e+0, esize], unsigned});
\]

\[
\text{op2} = \text{Int(Elem[operand, 2*e+1, esize], unsigned});
\]

\[
\text{sum} = (\text{op1+op2})<2*\text{esize}-1:0>;
\]

\[
\text{if acc then}
\]

\[
\text{Elem[result, e, 2*esize]} = \text{Elem[result, e, 2*esize]} + \text{sum};
\]

\[
\text{else}
\]

\[
\text{Elem[result, e, 2*esize]} = \text{sum};
\]

\[
V[d] = \text{result};
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09.rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register, while the SADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 0 | 0 | 0 | Rn | 0 | Rd | 1 |

SADDL\{2\} \<Vd>\,<Ta>, \<Vn>,\,<Tb>, \<Vm>\,<Tb>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in &quot;Q&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

\<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
\end{verbatim}

\textbf{Operational information}

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SADDLP

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 | Q | 0 | 0 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | Rd
  | U | op | Rn |
```

SADDLP `<Vd>.<Ta>`, `<Vn>.<Tb>`

integer d = `UInt`(Rd);
integer n = `UInt`(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << `UInt`(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');

**Assembler Symbols**

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th><code>&lt;Ta&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

`CheckFPAdvSIMDEnabled64();`

```c
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 | Q | 0 | 1 | 1 | 0 |  | size | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |  | Rd
  |   |   |   |   |   |   |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

SADDLV <V><d>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
   sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d] = sum<2*esize-1:0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The SADDW instruction extracts the second source vector from the lower half of the second source register, while the SADDW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

\[ \text{CheckFPAdvSIMDEnabled64}(); \]
\[
\text{bits}(2 \times \text{datasize}) \text{ operand1} = V[n];
\]
\[
\text{bits}(	ext{datasize}) \text{ operand2} = V\text{part}[m, \text{part}];
\]
\[
\text{bits}(2 \times \text{datasize}) \text{ result};
\]
\[
\text{integer element1};
\]
\[
\text{integer element2};
\]
\[
\text{integer sum};
\]
\[
\text{for } e = 0 \text{ to elements-1}
\]
\[
\quad \text{element1} = \text{Int}(\text{Elem}[\text{operand1}, e, 2 \times \text{esize}], \text{unsigned});
\]
\[
\quad \text{element2} = \text{Int}(\text{Elem}[\text{operand2}, e, \text{esize}], \text{unsigned});
\]
\[
\quad \text{if sub_op then}
\]
\[
\quad \quad \text{sum} = \text{element1} - \text{element2};
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{sum} = \text{element1} + \text{element2};
\]
\[
\quad \text{Elem}[\text{result}, e, 2 \times \text{esize}] = \text{sum}<2 \times \text{esize}-1:0>;
\]
\[
V[d] = \text{result};
\]

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SCVTF (vector, fixed-point)

Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | !=0000| 1   | 1   | 1   | 0   | 0   | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
U    |

SCVTF <V><d>, <V><n>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Vector

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 0  | !=0000| 1   | 1   | 1   | 0   | 0   | 1   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
U    |

SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh == '000x' || (immh == '001x' && !HaveFP16Ext()) then UNDEFINED;
if immm<3>:Q == '10' then UNDEFINED;
integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<V> Is a width specifier, encoded in "immh":

SCVTF (vector, fixed-point)
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>011x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>011x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding);

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: **Scalar half precision**, **Scalar single-precision and double-precision**, **Vector half precision** and **Vector single-precision and double-precision**

### Scalar half precision
(Armv8.2)

```
0 1 0 1 1 1 1 1 0 | 0 1 1 1 1 0 0 1 1 1 0 1 1 0 | Rn | Rd
```

SCVTF <Hd>, <Hn>

```java
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Scalar single-precision and double-precision

```
0 1 0 1 1 1 1 1 0 | 0 sz 1 0 0 0 0 1 1 1 0 1 1 0 | Rn | Rd
```

SCVTF <V><d>, <V><n>

```java
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector half precision
(Armv8.2)

```
0 0 0 1 1 1 0 | 0 1 1 1 1 0 0 1 1 1 0 1 1 0 | Rn | Rd
```

SCVTF (vector, integer)
SCVT <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SCVT <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

| 0 | 4H |
| 1 | 8H |

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

FPCRTYPE fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

FPRounding rounding = FPRoundingMode(fpcr);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| sf | 0 0 1 1 1 0 | ftype | 0 0 0 0 1 1 | scale | Rn | Rd |
```

sf: 0 (floating-point)
ftype: 0 (fixed-point)
scale: 1
Rn: 0
Rd: 0
rmode: 0
opcode: 1
32-bit to half-precision (sf == 0 && ftype == 11)
(Armv8.2)

SCVTF <Hd>, <Wn>, #<fbits>

32-bit to single-precision (sf == 0 && ftype == 00)

SCVTF <Sd>, <Wn>, #<fbits>

32-bit to double-precision (sf == 0 && ftype == 01)

SCVTF <Dd>, <Wn>, #<fbits>

64-bit to half-precision (sf == 1 && ftype == 11)
(Armv8.2)

SCVTF <Hd>, <Xn>, #<fbits>

64-bit to single-precision (sf == 1 && ftype == 00)

SCVTF <Sd>, <Xn>, #<fbits>

64-bit to double-precision (sf == 1 && ftype == 01)

SCVTF <Dd>, <Xn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  end case;
if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits> For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".
For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

**Operation**

```
CheckFPAdvSIMDEnabled64();
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, FALSE, fpcr, rounding);
V[d] = fltval;
```
SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see Floating-point exception traps.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
32-bit to half-precision (sf == 0 && ftype == 11)
(Armv8.2)

SCVTF <Hd>, <Wn>

32-bit to single-precision (sf == 0 && ftype == 00)

SCVTF <Sd>, <Wn>

32-bit to double-precision (sf == 0 && ftype == 01)

SCVTF <Dd>, <Wn>

64-bit to half-precision (sf == 1 && ftype == 11)
(Armv8.2)

SCVTF <Hd>, <Xn>

64-bit to single-precision (sf == 1 && ftype == 00)

SCVTF <Sd>, <Xn>

64-bit to double-precision (sf == 1 && ftype == 01)

SCVTF <Dd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTyperef pcr = FPCR[];
boolean merge = IsMerging(pcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem{fltval, 0, fltsize} = FixedToFP(intval, 0, FALSE, pcr, rounding);
V[d] = fltval;
SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_AA64ISARO_EL1.DP indicates whether this instruction is supported.

Vector
(Armv8.2)

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>L</th>
<th>M</th>
<th>Rm</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>H</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

SDOT <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.4B[<index>]

if !HaveDOTPExt() then UNDEFINED;
if size !='10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- `<Ta>` Is an arrangement specifier, encoded in “Q”:
  
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` Is an arrangement specifier, encoded in “Q”:
  
<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- `<index>` Is the element index, encoded in the "H:L" fields.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
b Reliable Simultaneous SIMD operations (AdvSIMD) result = V[d];
for e = 0 to elements-1
  integer res = 0;
  integer element1, element2;
  for i = 0 to 3
    if signed then
      element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
    else
      element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
      element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```
SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 0  | size| 0  | Rm | 1  | 0  | 0  | 1  | 0  | 1  | Rn | 1  | 0  | 0  | 1  | Rd |

SDOT <Vd>,<Ta>, <Vn>,<Tb>, <Vm>,<Tb>

if !HaveDOTPExt() then UNDEFINED;
if size != ’10’ then UNDEFINED;
    boolean signed = (U == ‘0’);
    integer d = UInt(Rd);
    integer n = UInt(Rn);
    integer m = UInt(Rm);
    integer esize = 8 << UInt(size);
    integer datasize = if Q == ’1’ then 128 else 64;
    integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the “Rd” field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];

bits(datasize) operand2 = V[m];

bits(datasize) result;

result = V[d];

for e = 0 to elements-1

    integer res = 0;
    integer element1, element2;

    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        
        res = res + element1 * element2;

        Elem[result, e, esize] = Elem[result, e, esize] + res;

V[d] = result;
SHA1C

SHA1 hash update (choose).

<table>
<thead>
<tr>
<th>SHA1C</th>
<th>&lt;Qd&gt;, &lt;Sn&gt;, &lt;Vm&gt;.4S</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer d = UInt(Rd);</td>
<td></td>
</tr>
<tr>
<td>integer n = UInt(Rn);</td>
<td></td>
</tr>
<tr>
<td>integer m = UInt(Rm);</td>
<td></td>
</tr>
<tr>
<td>if !HaveSHA1Ext() then UNDEFINED;</td>
<td></td>
</tr>
</tbody>
</table>

Assembler Symbols

- **<Qd>** Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
- **<Sn>** Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```plaintext
AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n];  // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;

for e = 0 to 3
    t = SHA1choose(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);

V[d] = X;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1H

SHA1 fixed rotate.

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SHA1H <Sd>, <Sn>

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Sd>  Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>  Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(32) operand = V[n];  // read element [0] only, [1-3] zeroed
V[d] = ROL(operand, 30);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1M

SHA1 hash update (majority).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SHA1M <Qd>, <Sn>, <Vm>.45

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n]; // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;
for e = 0 to 3
  t = SHAmajority(X<63:32>, X<95:64>, X<127:96>);
  Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
  X<63:32> = ROL(X<63:32>, 30);
  <Y, X> = ROL(Y:X, 32);
V[d] = X;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA1P

SHA1 hash update (parity).

|  31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  23  |  22  |  21  |  20  |  19  |  18  |  17  |  16  |  15  |  14  |  13  |  12  |  11  |  10  |  9   |  8   |  7   |  6   |  5   |  4   |  3   |  2   |  1   |  0   |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
|  0   |  1   |  0   |  1   |  1   |  1   |  1   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |  0   |

SHA1P <Qd>, <Sn>, <Vm>.45

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) X = V[d];
bits(32) Y = V[n];   // Note: 32 not 128 bits wide
bits(128) W = V[m];
bits(32) t;
for e = 0 to 3
    t = SHAparity(X<63:32>, X<95:64>, X<127:96>);
    Y = Y + ROL(X<31:0>, 5) + t + Elem[W, e, 32];
    X<63:32> = ROL(X<63:32>, 30);
    <Y, X> = ROL(Y:X, 32);
V[d] = X;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1SU0 schedule update 0.

<table>
<thead>
<tr>
<th>Rm</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0 0</td>
<td>0 0 1 1 0 0</td>
<td>0 1 1 0 0 0 0</td>
</tr>
</tbody>
</table>

SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
result = operand2<63:0>:operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA1SU1

SHA1 schedule update 1.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | |   |

SHA1SU1 <Vd>.4S, <Vn>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA256H2

SHA256 hash update (part 2).

|   31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  23  |  22  |  21  |  20  |  19  |  18  |  17  |  16  |  15  |  14  |  13  |  12  |  11  |  10  |  9   |  8   |  7   |  6   |  5   |  4   |  3   |  2   |  1   |  0   |
|-------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| 0     | 1    | 0    | 1    | 1    | 1    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    | 0    |

SHA256H2 <Qd>, <Qn>, <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[n], V[d], V[m], FALSE);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
SHA256H

SHA256 hash update (part 1).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

SHA256H <Qd>, <Qn>, <Vm>.

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.

<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[d], V[n], V[m], TRUE);
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA256SU0

SHA256 schedule update 0.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | Rn | Rd | Rn | Rd |

SHA256SU0 <Vd>.4S, <Vn>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();
bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) result;
bits(128) T = operand2<31:0>:operand1<127:32>;
bits(32) elt;
for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA256SU1

SHA256 schedule update 1.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |   |

SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d];
bits(128) operand2 = V[n];
bits(128) operand3 = V[m];
bits(128) result;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(64) T1;
bits(32) elt;
T1 = operand3<127:64>;
for e = 0 to 1
  elt = Elem[T1, e, 32];
  elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
  elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
  Elem[result, e, 32] = elt;
T1 = result<63:0>;
for e = 2 to 3
  elt = Elem[T1, e-2, 32];
  elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
  elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
  Elem[result, e, 32] = elt;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when \textit{FEAT\_SHA512} is implemented.

**Advanced SIMD**

\textit{(Armv8.2)}

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 2D |

SHA512H2 \(<Q_d>, <Q_n>, <V_m>\).2D

\[
\text{if } \neg \text{HaveSHA512Ext}() \text{ then UNDEFINED;}
\]

\[
\text{integer } d = \text{UInt}(Rd);
\]

\[
\text{integer } n = \text{UInt}(Rn);
\]

\[
\text{integer } m = \text{UInt}(Rm);
\]

**Assembler Symbols**

\(<Q_d>\) \quad \text{Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.}

\(<Q_n>\) \quad \text{Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.}

\(<V_m>\) \quad \text{Is the name of the third SIMD&FP source register, encoded in the "Rm" field.}

**Operation**

\texttt{AArch64.CheckFPAdvSIMDEnabled();}

\[
\begin{align*}
\text{bits(128)} & \quad V_{tmp}; \\
\text{bits(64)} & \quad N_{sigma0}; \\
\text{bits(128)} & \quad X = V[n]; \\
\text{bits(128)} & \quad Y = V[m]; \\
\text{bits(128)} & \quad W = V[d];
\end{align*}
\]

\[
\begin{align*}
N_{sigma0} & = \text{ROTR}(Y<63:0>, 28) \text{ EOR } \text{ROTR}(Y<63:0>, 34) \text{ EOR } \text{ROTR}(Y<63:0>, 39); \\
V_{tmp}<127:64> & = (X<63:0> \text{ AND } Y<127:64>) \text{ EOR } (X<63:0> \text{ AND } Y<63:0>) \text{ EOR } (Y<127:64> \text{ AND } Y<63:0>); \\
V_{tmp}<127:64> & = (V_{tmp}<127:64> + N_{sigma0} + W<127:64>); \\
N_{sigma0} & = \text{ROTR}(V_{tmp}<127:64>, 28) \text{ EOR } \text{ROTR}(V_{tmp}<127:64>, 34) \text{ EOR } \text{ROTR}(V_{tmp}<127:64>, 39); \\
V_{tmp}<63:0> & = (V_{tmp}<127:64> \text{ AND } Y<63:0>) \text{ EOR } (V_{tmp}<127:64> \text{ AND } Y<127:64>) \text{ EOR } (Y<127:64> \text{ AND } Y<63:0>); \\
V_{tmp}<63:0> & = (V_{tmp}<63:0> + N_{sigma0} + W<63:0>);
\end{align*}
\]

\[
V[d] = V_{tmp};
\]

**Operational information**

If \texttt{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  
  - The values of the data supplied in any of its registers.
  
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  
  - The values of the data supplied in any of its registers.
  
  - The values of the NZCV flags.
SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when `FEAT_SHA512` is implemented.

Advanced SIMD
(Armv8.2)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
| 1 1 0 0 1 1 1 0 0 1 1 | Rm | 1 0 0 0 0 0 | Rn | Rd |
```

SHA512H <Qd>, <Qn>, <Vm>.2D

if !HaveSHA512Ext() then UNDEFINED;
integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);

Assembler Symbols

<Qd> Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn> Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();
bits(128) Vtmp;
bits(64) MSigma1;
bits(128) tmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

MSigma1 = ROR(Y<127:64>, 14) EOR ROR(Y<127:64>, 18) EOR ROR(Y<127:64>, 41);
Vtmp<127:64> = (Y<127:64> AND X<63:0>) EOR (NOT(Y<127:64>) AND X<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + W<127:64>);
tmp = Vtmp<127:64> + Y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND Y<127:64>) EOR (NOT(tmp) AND X<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + W<63:0>);
V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when `FEAT_SHA512` is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |

SHA512SU0 <Vd>.2D, <Vn>.2D

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```assembly
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig0;
bites(128) Vtmp;
bites(128) X = V[n];
bites(128) W = V[d];
sig0 = ROR(W<127:64>, 1) EOR ROR(W<127:64>, 8) EOR ('0000000':W<127:71>);
Vtmp<63:0> = W<63:0> + sig0;
sig0 = ROR(X<63:0>, 1) EOR ROR(X<63:0>, 8) EOR ('0000000':X<63:7>);
Vtmp<127:64> = W<127:64> + sig0;
V[d] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when `FEAT_SHA512` is implemented.

Advanced SIMD
(Armv8.2)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 1 1 0 0 1 1</td>
</tr>
</tbody>
</table>

SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D

if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd>  Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>  Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>  Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

sig1 = ROR(X<127:64>, 19) EOR ROR(X<127:64>, 61) EOR ('000000':X<127:70>);
Vtmp<127:64> = W<127:64> + sig1 + Y<127:64>;
sig1 = ROR(X<63:0>, 19) EOR ROR(X<63:0>, 61) EOR ('000000':X<63:6>);
Vtmp<63:0> = W<63:0> + sig1 + Y<63:0>;
V[d] = Vtmp;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SRHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
○ The values of the data supplied in any of its registers.
○ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ○ The values of the data supplied in any of its registers.
  ○ The values of the NZCV flags.
**SHL**

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

**SHL** `<V>d`, `<V<n>`, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = UInt(immh:immb) - esize;
```

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

**SHL** `<Vd>.<T>`, `<Vn>.<T>`, #<shift>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = UInt(immh:immb) - esize;
```

### Assembler Symbols

- `<V>` Is a width specifier, encoded in “immh”:
  
<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.

- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- `<T>` Is an arrangement specifier, encoded in “immh:Q”:
<T>

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

**Operation**

```cpp
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHLL, SHLL2

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. The SHLL instruction extracts vector elements from the lower half of the source register, while the SHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

SHLL, SHLL2

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | Rd | Rd

SHLL(2) <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE; // Or TRUE without change of functionality

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8</td>
</tr>
<tr>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SHRN, SHRN2**

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see **RSHRN**.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| Q | 0 | 0 | 1 | 1 | 1 | 0 | implb | 1 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
```

<table>
<thead>
<tr>
<th>immh</th>
<th>op</th>
</tr>
</thead>
</table>

**Assembler Symbols**

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
</table>
0 | [absent] |
1 | [present] |
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”: 

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 1 | 0 | 0 | 1 | Rn | Rd |

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  diff = element1 - element2;
  Elem[result, e, esize] = diff<esize:1>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

The following figure shows an example of the operation of shift left by 3 for an 8-bit vector element.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```plaintext
SLI <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

Vector

```plaintext
SLI <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```
Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb) - 64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb) - 8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb) - 16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb) - 32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb) - 64)</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;
for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):ROL(Vm<95:64>, 15):ROL(Vm<63:32>, 15));
for i = 0 to 3
    if i == 3 then
        result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
    result<(32*i)+31:(32*i)> = result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>, 15) EOR ROL(result<(32*i)+31:(32*i)>, 23);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  |

SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):ROL(Vm<95:64>, 7):ROL(Vm<63:32>, 7):ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 1 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0
Rm 0 Ra 0 Rn Rd

SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va> Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(128) Va = V[a];
Vd<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
Vd<95:0> = Zeros();
V[d] = Vd;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ○ The values of the data supplied in any of its registers.
    ○ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ○ The values of the data supplied in any of its registers.
    ○ The values of the NZCV flags.
SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 0 1 1 1 0 0 1 0 | Rm | 1 0 | imm2 | 0 0 | Rn | Rd

SM3TT1A <Vd>.45, <Vn>.45, <Vm>.S[<imm2>]

if !HaveSM3Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when \texttt{FEAT_SM3} is implemented.

### Advanced SIMD (Armv8.2)

<table>
<thead>
<tr>
<th>Rm</th>
<th>Rd</th>
<th>imm2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0 1 0</td>
</tr>
</tbody>
</table>

\[
\text{SM3TT1B} \ <Vd>.4S, \ <Vn>.4S, \ <Vm>.S[\text{imm2}]
\]

if \(!\text{HaveSM3Ext}()\) then UNDEFINED;

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
integer i = \text{UInt}(\text{imm2});

### Assembler Symbols

- \(<Vd>\) is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
- \(<Vn>\) is the name of the second SIMD&FP source register, encoded in the "Rn" field.
- \(<Vm>\) is the name of the third SIMD&FP source register, encoded in the "Rm" field.
- \(<\text{imm2}>\) is a 32-bit element indexed out of \(<Vm>\), encoded in "imm2".

### Operation

\[
\text{AArch64.CheckFPAdvSIMDEnabled}();
\]

bits(128) Vm = \text{V}[m];
bits(128) Vn = \text{V}[n];
bits(128) Vd = \text{V}[d];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = \text{Elem}[Vm, i, 32];
SS2 = Vn<127:96> \text{EOR} \text{ROL}(Vd<127:96>, 12);
TT1 = (Vd<127:96> \text{AND} Vd<63:32>) \text{OR} (Vd<127:96> \text{AND} Vd<95:64>) \text{OR} (Vd<63:32> \text{AND} Vd<95:64>);
result<31:0> = Vd<63:32>;
result<63:32> = \text{ROL}(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d] = result;

### Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(Armv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>imm2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;
Wj = Elem[Vm, i, 32];
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
○ The values of the data supplied in any of its registers.
○ The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  ○ The values of the data supplied in any of its registers.
  ○ The values of the NZCV flags.
SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when FEAT_SM3 is implemented.

Advanced SIMD
(Armv8.2)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>imm2</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S<imm2>

if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2> Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) Vd = V[d];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = (Vd<127:96> AND Vd<95:64>) OR (NOT(Vd<127:96>) AND Vd<63:32>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register. This instruction is implemented only when FEAT_SM4 is implemented.

Advanced SIMD
(Armv8.2)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 0 1 1 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 1
Rn Rd

SM4E <Vd>.4S, <Vn>.4S

if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

Assembler Symbols

<Vd> Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn> Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vn = V[n];
bits(32) intval;
bits(8) sboxout;
bits(128) roundresult;
bits(32) roundkey;

roundkey = V[d];
for index = 0 to 3
    intval = Elem[Vn, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
V[d] = roundresult;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.

This instruction is implemented only when FEAT_SM4 is implemented.

Advanced SIMD
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  |

SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S

if !HaveSM4Ext() then UNDEFINED;
integer d = Uint(Rd);
integer n = Uint(Rn);
integer m = Uint(Rm);

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(32) intval;
bits(8) sboxout;
bits(128) result;
bits(32) const;
bits(128) roundresult;

roundresult = V[n];
for index = 0 to 3
  const = Elem[Vm, index, 32];
  intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
  for i = 0 to 3
    Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
  intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
  intval = intval EOR roundresult<31:0>;
roundresult<31:0> = roundresult<63:32>;
roundresult<63:32> = roundresult<95:64>;
roundresult<95:64> = roundresult<127:96>;
roundresult<127:96> = intval;
V[d] = roundresult;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem(operand1, e, esize), unsigned);
element2 = Int(Elem(operand2, e, esize), unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);  
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**SMAXP**

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | Rm | 1  | 0  | 1  | 0  | 0  | 1  | Rn | 0  | 1  | Rd |

**SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

**Assembler Symbols**

**<Vd>**

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>**

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>**

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Vm>**

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
  element1 = Int(Elem[concat, 2*e, esize], unsigned);
  element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
\end{verbatim}

### Assembler Symbols

- \(<V>\) Is the destination width specifier, encoded in “size”:

\begin{center}
\begin{tabular}{c|c}
\hline
size & \(<V>\) \\
\hline
00 & B \\
01 & H \\
10 & S \\
11 & RESERVED \\
\hline
\end{tabular}
\end{center}

- \(<d>\) Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

- \(<Vn>\) Is the name of the SIMD&FP source register, encoded in the “Rn” field.

- \(<T>\) Is an arrangement specifier, encoded in “size:Q”:

\begin{center}
\begin{tabular}{c|c|c}
\hline
size & Q & \(<T>\) \\
\hline
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & RESERVED \\
10 & 1 & 4S \\
11 & x & RESERVED \\
\hline
\end{tabular}
\end{center}

### Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
\end{verbatim}
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMIN

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
  element1 = Int(Elem(operand1, e, esize], unsigned);
  element2 = Int(Elem(operand2, e, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Q</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rd</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{CPTR\_EL3} registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\begin{verbatim}
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
\end{verbatim}

**Assembler Symbols**

\begin{verbatim}
<V> Is the destination width specifier, encoded in "size":
\begin{verbatim}
size <V>
00 B
01 H
10 S
11 RESERVED
\end{verbatim}

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in "size:Q":
\begin{verbatim}
size Q <T>
00 0 8B
00 1 16B
01 0 4H
01 1 8H
10 0 RESERVED
10 1 4S
11 x RESERVED
\end{verbatim}

**Operation**

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \texttt{V}[n];
integer maxmin;
integer element;
maxmin = \texttt{Int}(\texttt{Elem}[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = \texttt{Int}(\texttt{Elem}[operand, e, esize], unsigned);
    maxmin = if min then \texttt{Min}(maxmin, element) else \texttt{Max}(maxmin, element);
\texttt{V}[d] = maxmin<esize-1:0>;
\end{verbatim}
Operational information

If **PSTATE.DIT** is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register, while the SMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

### Assembler Symbols

- **2**
  - Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":
    ```
    Q 2
    0 [absent]
    1 [present]
    ```

- **<Vd>**
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Ta>**
  - Is an arrangement specifier, encoded in "size":
    ```
    size <Ta>
    00 RESERVED
    01 4S
    10 2D
    11 RESERVED
    ```

- **<Vn>**
  - Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

- **<Tb>**
  - Is an arrangement specifier, encoded in "size:Q":
    ```
    size Q <Tb>
    00 x RESERVED
    01 0 4H
    01 1 8H
    10 0 2S
    10 1 4S
    11 x RESERVED
    ```
Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
ingeger element1;
ingeger element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register, while the SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | Q   | 0   | 0   | 1   | 1   | 1   | 0   | size| 1   | Rm  | 1   | 0   | 0   | 0   | 0   | Rn  | Rd  |
| U   | o1  |
```

**SMLAL[2]** <Vd>, <Ta>, <Vn>, <Tb>, <Vm>, <Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>0</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
     ◦ The values of the data supplied in any of its registers.
     ◦ The values of the NZCV flags.
   • The response of this instruction to asynchronous exceptions does not vary based on:
     ◦ The values of the data supplied in any of its registers.
     ◦ The values of the NZCV flags.
SMLSL, SMLSL2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts vector elements from the lower half of the first source register, while the SMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| U | Q | 0 | 0 | 1 | 1 | 1 | size | L | M | Rm | 0 | 1 | 1 | 0 | H | 0 | Rn | Rd | o2 |

SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>45</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size.Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
   element1 = Int(Elem[operand1, e, esize], unsigned);
   product = (element1*element2)<2*esize-1:0>;
   if sub_op then
      Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
   else
      Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register, while the SMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|--------|--------|--------|-------|--------|-------|--------|--------|
|   U    |   Q    |   size|   Rd  |   Rm   |   Rn   |   Rd   |
| 0 0 0 1 1 0 | 1 0 1 0 | 0 0   |
```

SMLSL\{2\} <Vd>., <Ta>, <Vn>., <Tb>.<Tb>

```plaintext
d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

if size == '11' then UNDEFINED;
esize = 8 << UInt(size);
datasize = 64;
part = UInt(Q);
elements = datasize DIV esize;
sub_op = (o1 == '1');
unsigned = (U == '1');
```

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
<table>
<thead>
<tr>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 [absent]</td>
</tr>
<tr>
<td>1 [present]</td>
</tr>
</tbody>
</table>
```

<Vd> is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> is an arrangement specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> is the name of the first SIMD&FP source register, encoded in the “Rn” field.

<Tb> is an arrangement specifier, encoded in “size:Q”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vm> is the name of the second SIMD&FP source register, encoded in the “Rm” field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMMLA (vector)

Signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an **OPTIONAL** instruction.  

ID_AA64ISAR0_EL1.I8MM indicates whether this instruction is supported.

Vector  
(Armv8.6)

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------|----------------|----------------|----------------|
|          |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |       |
| 0 1 0 0 1 1 1 0 1 0 0 | Rm | 1 0 1 0 0 1 | Rn | Rd | B |
```

SMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```plaintext
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];
V[d] = MatMulAdd(addend, operand1, operand2, FALSE, FALSE);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMOV

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

32-bit (Q == 0)

SMOV <Wd>, <Vn>.<Ts>[<index>]

64-reg,SMOV-64-reg (Q == 1)

SMOV <Xd>, <Vn>.<Ts>[<index>]

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
  when 'xxxxx1' size = 0; // SMOV [WX]d, Vn.B
  when 'xxxx10' size = 1; // SMOV [WX]d, Vn.H
  when '1xx100' size = 2; // SMOV Xd, Vn.S
  otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
</tbody>
</table>

For the 64-reg,SMOV-64-reg variant: is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
</tbody>
</table>

[index>] For the 32-bit variant: is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
</tbody>
</table>

For the 64-reg,SMOV-64-reg variant: is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx10</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
</tbody>
</table>
### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
```

```c
X[d] = SignExtend(Elem[operand, index, esize], datasize);
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SMULL, SMULL2 (by element)**

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register, while the SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | Q  | 0  | 1  | 1  | 1  | size | L | M | Rm | 1  | 0  | 1  | 0  | H  | 0  | Rn | 1  | 0  | 1  | 0  | 1  | 0  | H  | 0  | Rn | 1  | 0  | 1  | 0  | Rm |

**SMULL{2} \(<Vd>,<Ta>,<Vn>,<Tb>,<Vm>[<index>]\)**

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<Ta>\) Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;Ta&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Tb>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>(&lt;Tb&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”: 
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULL, SMULL2 (vector)

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register, while the SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q   | 0  | 0  | 1  | 1  | 0  | size | 1  | Rm | 1  | 1  | 0  | 0  | 0  | Rn |   | Rd |

SMULL2 {2} <Vd>, <Ta>, <Vn>, <Vm>, <Tb>
```

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

**Assembler Symbols**

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SQABS

Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | Rd |
| U |

SQABS <V><d>, <V><n>

```plaintext
d = UInt(Rd);
n = UInt(Rn);
esize = 8 << UInt(size);
datasize = esize;
elements = 1;
neg = (U == '1');
```

**Vector**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | Q | 0 | 1 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | Rd |
| U |

SQABS <Vd>.<T>, <Vn>.<T>

```plaintext
d = UInt(Rd);
n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
esize = 8 << UInt(size);
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV esize;
neg = (U == '1');
```

**Assembler Symbols**

<

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q".
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
SQADD

Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 1 0 1 1 1 0    | size 1           | Rm 0 0 0 0 1 1   | Rn               |
| U                |                  | Rd              |
```

SQADD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0 0 1 1 1 0      | size 1           | Rm 0 0 0 0 1 1   | Rn               |
| U                |                  | Rd              |
```

SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

```
<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<\text{n}> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\text{m}> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;
for e = 0 to elements-1
  element1 = \text{Int}(\text{Elem}[operand1, e, esize], unsigned);
  element2 = \text{Int}(\text{Elem}[operand2, e, esize], unsigned);
  sum = element1 + element2;
  (\text{Elem}[result, e, esize], sat) = \text{SatQ}(sum, esize, unsigned);
  if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMLAL, SQDMLAL2 (by element)**

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set.

The $SQDMLAL$ instruction extracts vector elements from the lower half of the first source register, while the $SQDMLAL2$ instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
```

```
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
```

```
ingeter esize = 8 << UInt(size);
ingeter datasize = esize;
ingeter elements = 1;
ingeneter part = 0;
```

```
boolean sub_op = (o2 == '1');
```

### Vector
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper
64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:
Restricted to V0-V15 when element size $<Ts>$ is H.

$<Ts>$ is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;Vm&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<index>$ is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;index&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
```
SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLAL instruction extracts each source vector from the lower half of each source register, while the SQDMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

![Scalar Encoding](image)

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');

**Vector**

![Vector Encoding](image)

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the source width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
```
SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLSL instruction extracts vector elements from the lower half of the first source register, while the SQDMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQDMLSL <Va><d>, <Vb><n>, <Vm><Ts>[<index>]

integer idxdsiz = if H == '1' then 128 else 64;
integer index;
bit Rmhi;

---

**Vector**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>size</td>
<td>L</td>
<td>M</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>H</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td>o2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQDMLSL, SQDMLSL2 (by element)
integer idxsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”: 
Restricted to V0-V15 when element size $<Ts>$ is H.

$<Ts>$ is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<\text{index}>$ is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\text{index}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMLSL, SQDMLSL2 (vector)**

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMLSL instruction extracts each source vector from the lower half of each source register, while the SQDMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rm</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 0</td>
<td>1</td>
<td>0 1 1 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SQDMLSL <Va><d>, <Vb><n>, <Vb><m>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rm</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0 1 1 1 0</td>
<td>1</td>
<td>0 1 1 0 0</td>
</tr>
</tbody>
</table>

**SQDMLSL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');

### Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bias(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';
V[d] = result;
```

SQDMLSL, SQDMLSL2 (vector)
**SQDMULH (by element)**

Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see SQRDMLULH.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | L  | M  | Rm | 1  | 1  | 0  | 0  | H  | 0  | Rn | Rd |
```

\[
\text{SQDMULH} <V><d>, <V><n>, <Vm>.<Ts>[<index>]
\]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');

**Vector**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | L  | M  | Rm | 1  | 1  | 0  | 0  | H  | 0  | Rn | Rd |
```

\[
\text{SQDMULH} <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]
\]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index> Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

\texttt{CheckFPAdvSIMDEnabled64}();

\texttt{bits(datasize) operand1 = V[n];}
\texttt{bits(idxdsize) operand2 = V[m];}
\texttt{bits(datasize) result;}
\texttt{integer round\_const = if round then 1 \ll (esize - 1) else 0;}
\texttt{integer element1;}
\texttt{integer element2;}
\texttt{integer product;}
\texttt{boolean sat;}

\texttt{element2 = \texttt{SInt}(Elem[operand2, index, esize]);}
\texttt{for e = 0 to elements-1}
\texttt{element1 = \texttt{SInt}(Elem[operand1, e, esize]);}
\texttt{product = (2 \ast element1 \ast element2) + round\_const;}
\texttt{// The following only saturates if element1 and element2 equal -(2^{(esize-1)})}
\texttt{(Elem[result, e, esize], sat) = \texttt{SignedSatQ}(product \gg esize, esize);}
\texttt{if sat then FPSR.QC = '1';}

\texttt{V[d] = result;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMULH (vector)**

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *SQRDMULH*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  1  0  1  1  1  1  0  | size  |  1  |  Rm  |  1  0  1  1  0  1  |  Rn  |  Rd

U
```

SQDMULH <V<d>, <V<n>, <V<m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');

### Vector

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  1  1  1  0  | size  |  1  |  Rm  |  1  0  1  1  0  1  |  Rn  |  Rd

U
```

SQDMULH <Vd>,.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');

### Assembler Symbols

- `<V>` Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
- `<n>` Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- `<m>` Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```pseudo
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round Const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round Const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

The SQDMULL instruction extracts the first source vector from the lower half of the first source register, while the SQDMULL2 instruction extracts the first source vector from the upper half of the first source register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 size L M Rm 1 0 1 1 H 0 Rn Rd

SQDMULL <Va><d>, <Vb><n>, <Vm><Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

Vector

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 1 1 1 size L M Rm 1 0 1 1 H 0 Rn Rd

SQDMULL[2] <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
Assembler Symbols

2
Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>
Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Va>
Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb>
Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>
Is the name of the second SIMD&FP source register, encoded in "size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts>
Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<index>
Is the element index, encoded in "size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d] = result;
**SQDMULL, SQDMULL2 (vector)**

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register, while the SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rm</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>1</td>
<td>1 0 1 0 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>

SQDMULL $<Va><d>$, $<Vb><n>$, $<Vb><m>$

integer d = $UInt(Rd)$;
integer n = $UInt(Rn)$;
integer m = $UInt(Rm)$;

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 $<< UInt(size)$;
integer datasize = esize;
integer elements = 1;
integer part = 0;

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rm</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 0 1 1 1 0</td>
<td>1</td>
<td>1 0 1 0 0</td>
<td>Rn</td>
</tr>
</tbody>
</table>

SQDMULL2 $<Vd>.<Ta>$, $<Vn>.<Tb>$, $<Vm>.<Tb>$

integer d = $UInt(Rd)$;
integer n = $UInt(Rn)$;
integer m = $UInt(Rm)$;

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 $<< UInt(size)$;
integer datasize = 64;
integer elements = datasize DIV esize;

### Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in &quot;Q&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<Ta>$ Is an arrangement specifier, encoded in “size”: 

SQDMULL, SQDMULL2 (vector)
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is an arrangement specifier, encoded in "size:Q":

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Is the destination width specifier, encoded in "size":

Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

Is the source width specifier, encoded in "size":

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';
V[d] = result;
```
Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 0 0 0 0 0 1 1 1 1 0</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
</table>

u

**SQNEG** <V><d>, <V><n>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 0 0 0 0 0 0 0 1 1 1 1 0</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
</table>

u

**SQNEG** <Vd>..<T>, <Vn>..<T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');

### Assembler Symbols

**<V>** is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**<d>** is the number of the SIMD&FP destination register, encoded in the "Rd" field.

**<n>** is the number of the SIMD&FP source register, encoded in the "Rn" field.

**<Vd>** is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<T>** is an arrangement specifier, encoded in “size:Q”:
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLAH (by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(Armv8.1)

Scalar <V><d>, <V><n>, <Vm>.<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector
(Armv8.1)

Vector <V><O>, <V><O>, <Vm>.<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
SQRDMLAH \(<Vd>\), <T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

if !\text{HaveSQRDMLAHExt}() then UNDEFINED;

integer idxdsize = if \(H == '1'\) then 128 else 64;
integer index;
bit Rmhi;
\text{case size of}
\hspace{1em} when '01' index = \text{UInt}(H:L:M); Rmhi = '0';
\hspace{1em} when '10' index = \text{UInt}(H:L); Rmhi = M;
\hspace{1em} otherwise UNDEFINED;

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rmhi:Rm);

integer esize = 8 \ll \text{UInt}(size);
integer datasize = if \(Q == '1'\) then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

\textbf{Assembler Symbols}

\(<V>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<n>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

\(<T>\) Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size \(<Ts>\) is H.

\(<Ts>\) Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<index>\) Is the element index, encoded in “size:L:H:M”:
**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(Armv8.1)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

SQRDMLAH <V><d>, <V><n>, <V><m>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector
(Armv8.1)

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
</tr>
</tbody>
</table>

SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```
SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(Armv8.1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1

SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;

case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector
(Armv8.1)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1

SQRDMLSH (by element)
SQRDMLSH $<Vd>,<T>,<Vn>,<T>,<Vm>,<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

$<V>$ Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<d>$ Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

$<n>$ Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

$<Vm>$ Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size $<Ts>$ is H.

$<Ts>$ Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<index>$ Is the element index, encoded in “size:L:H:M”:
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded. If any of the results overflow, they are saturated. The cumulative saturation bit, FPSR.QC, is set if saturation occurs. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar
(Armv8.1)

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & \text{size} & 0 & \text{Rm} & 1 & 0 & 0 & 0 & 1 & 1 & \text{Rn} & \text{Rd} & S
\end{array}
\]

SQRDMMLSH <V>d>, <V>n>, <V>m>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == 'll' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Vector
(Armv8.1)

\[
\begin{array}{cccccccccccc}
0 & 0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & \text{size} & 0 & \text{Rm} & 1 & 0 & 0 & 0 & 1 & 1 & \text{Rn} & \text{Rd} & S
\end{array}
\]

SQRDMMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == 'll' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
boolean sat;
for e = 0 to elements-1
   element1 = SInt(Elem[operand1, e, esize]);
   element2 = SInt(Elem[operand2, e, esize]);
   element3 = SInt(Elem[operand3, e, esize]);
   if sub_op then
      accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
   else
      accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
   (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
   if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQDMULH. If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 1 0 1 1 1 1 1 size</th>
<th>L M</th>
<th>Rm</th>
<th>1 1 0 1 H 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

SQRDMULH <V>d>, <V>n>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 | Q | 0 | 0 | 1 | 1 | 1 | size | L | M | Rm | 1 | 1 | 0 | 1 | H | 0 | Rn | Rd |
|-----------------------------------------------|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|----|-----|----|----|-----|----|----|-----|

SQRDMULH <Vd>, <T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
## Assembler Symbols

### <V>
Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### <d>
Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

### <n>
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

### <Vd>
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

### <T>
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### <Vn>
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

### <Vm>
Is the name of the second SIMD&FP source register, encoded in "size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

### <Ts>
Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### <index>
Is the element index, encoded in “size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```
SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SQRDMULH.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rm</th>
<th>1 0 1 1 0 1 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U SQRDMULH &lt;V&gt;&lt;d&gt;, &lt;V&gt;&lt;n&gt;, &lt;V&gt;&lt;m&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

type d = UInt(Rd);
type n = UInt(Rn);
type m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
type esize = 8 << UInt(size);
type datasize = esize;
type elements = 1;
boolean rounding = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rm</th>
<th>1 0 1 1 0 1 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U SQRDMULH &lt;Vd&gt;.&lt;T&gt;, &lt;Vn&gt;.&lt;T&gt;, &lt;Vm&gt;.&lt;T&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

type d = UInt(Rd);
type n = UInt(Rn);
type m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
type esize = 8 << UInt(size);
type datasize = if Q == '1' then 128 else 64;
type elements = datasize DIV esize;
boolean rounding = (U == '1');

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>01</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

| <d> | Is the number of the SIMD&FP destination register, in the "Rd" field. |
| <n> | Is the number of the first SIMD&FP source register, encoded in the "Rn" field. |
| <m> | Is the number of the second SIMD&FP source register, encoded in the "Rm" field. |
| <Vd> | Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see SQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 0  | 1  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U  | R  | S  |

SQRSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' &
size != '11' then UNDEFINED;

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | 0  | 1  | 1  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| U  | Q  | R  | S  |

SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
        if saturating then
            (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
            if sat then FPSR.QC = '1';
        else
            Elem[result, e, esize] = element<esize-1:0>;
    
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see SQRSHRN.

The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

### Scalar

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 1  | 0  | 0  | 1  | 1  | Rn  | Rd  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SQRSHRN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

### Vector

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |    |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | 0  | Q  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 1  | 0  | 0  | 1  | 1  | Rn  | Rd  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SIE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

\[
\begin{array}{c|c}
Q & 2 \\
0 & \text{[absent]} \\
1 & \text{[present]} \\
\end{array}
\]

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.

<Va> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-\text{UInt}(\text{immh}:\text{immb}))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-\text{UInt}(\text{immh}:\text{immb}))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-\text{UInt}(\text{immh}:\text{immb}))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-\text{UInt}(\text{immh}:\text{immb}))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-\text{UInt}(\text{immh}:\text{immb}))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-\text{UInt}(\text{immh}:\text{immb}))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```
SQRSHRUN, SQRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see SQRSHRUN.

The SQRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 1  | 0  | 0  | 0  | 1  | 1  | Rn  | Rd  |

SQRSHRUN <Vb><d>, <Va><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');

Vector

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | != 0000 | immh | 1  | 0  | 0  | 0  | 1  | 1  | Rn  | Rd  |

SQRSHRUN{2} <Vd>..<Tb>, <Vn>..<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

SQRSHRUN, SQRSHRUN2
Q

<table>
<thead>
<tr>
<th></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Ta</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Vb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Va</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements - 1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSHL (immediate)**

Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see **UQRSHL**.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|  31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  23  |  22  |  21  |  20  |  19  |  18  |  17  |  16  |  15  |  14  |  13  |  12  |  11  |  10  |  9   |  8   |  7   |  6   |  5   |  4   |  3   |  2   |  1   |  0   |
|  0   |  1   |  1   |  1   |  1   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|     ! = 0000       |      immh         |        Rn         |        Rd         |
+-------------------+-------------------+-------------------+-------------------+
|      U            |                   |                   |                   |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
```

**SQSHL** `<V><d>`, `<V><cn>`, `<shift>`

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

### Vector

```
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|  31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  23  |  22  |  21  |  20  |  19  |  18  |  17  |  16  |  15  |  14  |  13  |  12  |  11  |  10  |  9   |  8   |  7   |  6   |  5   |  4   |  3   |  2   |  1   |  0   |
|  0   |  0   |  0   |  1   |  1   |  1   |  1   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  0   |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|     ! = 0000       |      immh         |        Rn         |        Rd         |
+-------------------+-------------------+-------------------+-------------------+
|      U            |                   |                   |                   |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
```

**SQSHL** `<Vd>.<T>`, `<Vn>.<T>`, `<shift>`

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then **SEE(asimdimm)**;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>01x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0 8B</td>
</tr>
<tr>
<td>0001</td>
<td>1 16B</td>
</tr>
<tr>
<td>01x</td>
<td>0 4H</td>
</tr>
<tr>
<td>01x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>01x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>01x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
  element = Int(Elem[operand, e, esize], src_unsigned) << shift;
  (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
  if sat then FPSR.QC = '1';
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSHL (register)**

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see **SQRSHL**.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1, CPACR_EL2**, and **CPACR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0   1   1   1   1 0 | size 1 | Rm 0 1 0 0 1 1 | Rd |
| U   R   S          |       |                |                |
```

**SQSHL** `<V>`<d>, `<V>`<n>, `<V>`<m>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- integer esize = 8 << UInt(size);
- integer datasize = esize;
- integer elements = 1;
- boolean unsigned = (U == '1');
- boolean rounding = (R == '1');
- boolean saturating = (S == '1');
- if S == '0' && size != '11' then UNDEFINED;

**Vector**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|
| 0   Q 0 1 1 1 1 0 | size 1 | Rm 0 1 0 0 1 1 | Rd |
| U   R   S          |       |                |                |
```

**SQSHL** `<V>`<d>,`<T>`, `<V>`<n>,`<T>`, `<V>`<m>,`<T>`

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size:Q == '110' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- boolean unsigned = (U == '1');
- boolean rounding = (R == '1');
- boolean saturating = (S == '1');

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQSHLU

Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see **UQRSHL**.

If saturation occurs, the cumulative saturation bit **FPSR.QC** is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>! = 0000</th>
<th>immh</th>
<th>0 1 1 0 0 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>( \text{imm} )</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQSHLU \(<V>d\>, \(<V>n\>, \#<shift>

integer \(d\) = \(\text{UInt}(Rd)\);
integer \(n\) = \(\text{UInt}(Rn)\);

if \(\text{immh} == '0000'\) then UNDEFINED;
integer \(\text{esize} = 8 << \text{HighestSetBit}(\text{immh})\);
integer \(\text{datasize} = \text{esize}\);
integer \(\text{elements} = 1\);

integer \(\text{shift} = \text{UInt}(\text{immh}:\text{immb}) - \text{esize}\);

boolean \(\text{src\_unsigned}\);
boolean \(\text{dst\_unsigned}\);
case \(\text{op}:U\) of
  when '00' UNDEFINED;
  when '01' \(\text{src\_unsigned} = \text{FALSE}; \text{dst\_unsigned} = \text{TRUE}\);
  when '10' \(\text{src\_unsigned} = \text{FALSE}; \text{dst\_unsigned} = \text{FALSE}\);
  when '11' \(\text{src\_unsigned} = \text{TRUE}; \text{dst\_unsigned} = \text{TRUE}\);

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>! = 0000</th>
<th>immh</th>
<th>0 1 1 0 0 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>( \text{imm} )</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQSHLU \(<Vd>.,\langle T\rangle, \langle Vn\rangle.,\langle T\rangle, \#<shift>

integer \(d\) = \(\text{UInt}(Rd)\);
integer \(n\) = \(\text{UInt}(Rn)\);

if \(\text{immh} == '0000'\) then \(\text{SEE(asimdimm)}\);
if \(\text{immh}>3:x == '10'\) then UNDEFINED;
integer \(\text{esize} = 8 << \text{HighestSetBit}(\text{immh})\);
integer \(\text{datasize} = \text{if } Q == '1' \text{ then 128 else 64} ;\)
integer \(\text{elements} = \text{datasize DIV esize} ;\)

integer \(\text{shift} = \text{UInt}(\text{immh}:\text{immb}) - \text{esize} ;\)

boolean \(\text{src\_unsigned}\);
boolean \(\text{dst\_unsigned}\);
case \(\text{op}:U\) of
  when '00' UNDEFINED;
  when '01' \(\text{src\_unsigned} = \text{FALSE}; \text{dst\_unsigned} = \text{TRUE}\);
  when '10' \(\text{src\_unsigned} = \text{FALSE}; \text{dst\_unsigned} = \text{FALSE}\);
  when '11' \(\text{src\_unsigned} = \text{TRUE}; \text{dst\_unsigned} = \text{TRUE}\);
**Assembler Symbols**

### <V>

Is a width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

### <d>

Is the number of the SIMD&FP destination register, in the "Rd" field.

### <n>

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

### <Vd>

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

### <T>

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0 8B</td>
</tr>
<tr>
<td>0001</td>
<td>1 16B</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

### <Vn>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

### <shift>

For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSHRN, SQSHRN2**

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see **SQSHRN**.

The SQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit **FPSR**.QC is set.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 1   | 1   | 1   | 1   | 0   | != 0000 | immh | 1   | 0   | 0   | 1   | 0   | 1   | Rn | Rd |

U

immh

op

SQSHRN <Vb><d>, <Va><n>, #<shift>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

### Vector

```
| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 1   | 1   | 1   | 1   | 0   | != 0000 | immh | 1   | 0   | 0   | 1   | 0   | 1   | Rn | Rd |

U

immh

op

SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SFE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```
Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>

Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Tb>

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Ta>

Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb>

Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d>

Is the number of the SIMD&FP destination register, in the “Rd” field.

<Va>

Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n>

Is the number of the first SIMD&FP source register, encoded in the “Rn” field.

<shift>

For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
if sat then FPSR.QC = '1';
Vpart[d, part] = result;
SQSHRUN, SQSHRUN2

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see SQRSRUN.

The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>!= 0000</th>
<th>immh</th>
<th>1 0 0 0</th>
<th>0</th>
<th>1</th>
<th>Rd</th>
<th>Rn</th>
<th>op</th>
<th>immh</th>
</tr>
</thead>
</table>

SQSHRUN <Vb><d>, <Va><n>, #<shift>

```plaintext
d = UINT(Rd);
n = UINT(Rn);
if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
esize = 8 << HighestSetBit(immh);
dataSize = esize;
elements = 1;
opPart = 0;
shift = (2 * esize) - UINT(immh:immb);
round = (op == '1');
```

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>!= 0000</th>
<th>immh</th>
<th>1 0 0 0</th>
<th>0</th>
<th>1</th>
<th>Rd</th>
<th>Rn</th>
<th>op</th>
</tr>
</thead>
</table>

SQSHRUN[2] <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```plaintext
d = UINT(Rd);
n = UINT(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
esize = 8 << HighestSetBit(immh);
dataSize = 64;
opPart = UINT(Q);
elements = dataSize DIV esize;
shift = (2 * esize) - UINT(immh:immb);
round = (op == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```plaintext
2
```
**Q**

<table>
<thead>
<tr>
<th></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

**<Vd>**
Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Tb>**
Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
</tr>
</tbody>
</table>

**<Vn>**
Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<Ta>**
Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vb>**
Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<d>**
Is the number of the SIMD&FP destination register, in the "Rd" field.

**<Va>**
Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<n>**
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

**<shift>**
For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>001x</td>
<td>(32-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>001x</td>
<td>(32-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-$\text{UInt}(\text{immh:immb})$)</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
  element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
  (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
  if sat then FPSR.QC = '1';
Vpart[d, part] = result;
SQSUB

Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

SQSUB <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQXTN, SQXTN2**

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \( FPSR.QC \) is set.

The SQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the \( CPACR_EL1, CPTR_EL2, \) and \( CPTR_EL3 \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 1   | 1   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 1   |

\( Rn \) \( Rd \)

**U**

\[ \text{SQXTN} \ <Vb><d>, \ <Va><n> \]

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
boolean unsigned = (U == '1');
```

### Vector

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 1   | 1   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 1   |

\( Rn \) \( Rd \)

**U**

**SQXTN(2) \ <Vd>.<Tb>, \ <Vn>.<Ta>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

### Assembler Symbols

2

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

**<Vd>**

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 8H</td>
</tr>
<tr>
<td>01 4S</td>
</tr>
<tr>
<td>10 2D</td>
</tr>
<tr>
<td>11 RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 B</td>
</tr>
<tr>
<td>01 H</td>
</tr>
<tr>
<td>10 S</td>
</tr>
<tr>
<td>11 RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 H</td>
</tr>
<tr>
<td>01 S</td>
</tr>
<tr>
<td>10 D</td>
</tr>
<tr>
<td>11 RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set.

The `SQXTUN` instruction writes the vector to the lower half of the destination register and clears the upper half, while the `SQXTUN2` instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | | |
| Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

`SQXTUN <Vb><d>, <Va><n>`

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
```

**Vector**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | | |
| Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

`SQXTUN{2} <Vd>, <Tb>, <Vn>, <Ta>`

```plaintext```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

`<Tb>` Is an arrangement specifier, encoded in “size:Q”:
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<size>Q</size> <Ta>

Is an arrangement specifier, encoded in "size":

<size> <Ta>

<table>
<thead>
<tr>
<th>size</th>
<th>00</th>
<th>8H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in "size":

<size> <Vb>

<table>
<thead>
<tr>
<th>size</th>
<th>00</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in "size":

<size> <Va>

<table>
<thead>
<tr>
<th>size</th>
<th>00</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see SHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|--------|--------|--------|--------|
| size Q Rm | 0 0 1 1 0 | Rd | 0 0 0 1 0 1 | Rn | Rd |

SRHADD <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>;
V[d] = result;
Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost. 

The following figure shows an example of the operation of shift right by 3 for an 8-bit vector element.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | != | 0000 | immh | 0 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |

SRI \(<V\><d>, \langle V\rangle<\langle T\rangle, \langle Vn\rangle<\langle T\rangle, \#<shift>

integer d = UInt(Rd);  
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;  
integer esize = 8 << 3;  
integer datasize = esize;  
integer elements = 1;  
integer shift = (esize * 2) - UInt(immh:immb);

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | != | 0000 | immh | 0 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |

SRI \(<Vd>.\langle T\rangle, \langle Vn\>.\langle T\rangle, \#<shift>

integer d = UInt(Rd);  
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);  
if immh<3>:Q == '10' then UNDEFINED;  
integer esize = 8 << HighestSetBit(immh);  
integer datasize = if Q == '1' then 128 else 64;  
integer elements = datasize DIV esize;  
integer shift = (esize * 2) - UInt(immh:immb);
Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0 8B</td>
</tr>
<tr>
<td>0001</td>
<td>1 16B</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2 = V[d];
bits(datasize) result;
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shifted;
for e = 0 to elements-1
    shifted = LSR(Elem(operand, e, esize), shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see `SSHL`. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
Scalar  <V><d>, <V><n>, <V><m>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

### Vector

```
Vector  <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

### Assembler Symbols

<table>
<thead>
<tr>
<th><code>&lt;V&gt;</code></th>
<th>Is a width specifier, encoded in &quot;size&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size</code></td>
<td><code>&lt;V&gt;</code></td>
</tr>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

| `<d>` | Is the number of the SIMD&FP destination register, in the "Rd" field. |
| `<n>` | Is the number of the first SIMD&FP source register, encoded in the "Rn" field. |
| `<m>` | Is the number of the second SIMD&FP source register, encoded in the "Rm" field. |
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;
for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1);    // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SRSHR

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSSHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

SRSHR <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

SRSHR <Vd><T>, <Vn><T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_Advanced_SIMDModified_Immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMDModified_Immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see SSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 1 1 0</td>
</tr>
</tbody>
</table>

U

immh

o1 o0

SRSRA <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

U

immh

o1 o0

SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm == '0000' then ssra(asmimm);
if immh<3>:Q == '01' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Assembler Symbols**

<V>  Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d>  Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see SRSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

SSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0</td>
</tr>
</tbody>
</table>

SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
  shift = SInt(Elem[operand2, e, esize]<7:0>);
  if rounding then
    round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
  element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
  if saturating then
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
  else
    Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SSHLL, SSHLL2**

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register, while the SSHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias `SXTL, SXTL2`.

![Table](image)

**Assemble Symbols**

2  

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta>  

Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVE</td>
</tr>
</tbody>
</table>

<Vn>  

Is the name of the SIMD&FP source register, encoded in the “Rn” field.

<Tb>  

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001 0</td>
<td>8B</td>
</tr>
<tr>
<td>0001 1</td>
<td>16B</td>
</tr>
<tr>
<td>001x 0</td>
<td>4H</td>
</tr>
<tr>
<td>001x 1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx 0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx 1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx x</td>
<td>RESERVE</td>
</tr>
</tbody>
</table>

<shift>  

Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;

integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
<table>
<thead>
<tr>
<th>immh</th>
<th>&quot;&lt;shift&gt;&quot;</th>
<th>SEE Advanced SIMD modified immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td></td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td></td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>SXTL, SXTL2</td>
<td>immh == '000' &amp; BitCount(immh) == 1</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;
V[d] = result;
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SSHR**

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see **SRSHR**.

Depending on the settings in the **CPACR_EL1**, **CPR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 1 1 1 1 1 1 0 | != 0000 | immh | 0 0 0 0 0 1 | Rd |
| U | o1 o0 |

**SSHR <V><d>, <V><n>, #<shift>**

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Vector**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | != 0000 | immh | 0 0 0 0 0 1 | Rd |
| U | o1 o0 |

**SSHR <Vd>,<T>, <Vn>,<T>, #<shift>**

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then **SEE(asimdimm)**;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << **HighestSetBit**(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

**Assembler Symbols**

**<V>**

Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

**<d>**

Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];

bits(datasize) operand2;

bits(datasize) result;

integer round_const = if round then (1 << (shift - 1)) else 0;

integer element;

operand2 = if accumulate then V[d] else Zeros();

for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see SRSRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 1 1 1 1 1 1 0 != 0000 immb 0 0 0 1 0 1 Rn Rd
U immh 01 00
```

SSRA <V><d>, <V><n>, ##<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if imm<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 1 1 1 0 != 0000 immb 0 0 0 1 0 1 Rn Rd
U immh 01 00
```

SSRA <Vd>,<T>, <Vn>.<T>, ##<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if imm == '0000' then SEE(asimdimm);
if imm<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<\textbf{V}> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>\textbf{V}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0 8B</td>
</tr>
<tr>
<td>0001</td>
<td>1 16B</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1 2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEEAdvancedSIMDmodifiedImmediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register, while the SSUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
SSUBL2<VD>,<TA>,<VN>,<TB>>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<VD> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<TA> Is an arrangement specifier, encoded in “size”:

```

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;TA&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<VN> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<TB> Is an arrangement specifier, encoded in “size:Q”:

```

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;TB&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<VM> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**SSUBW, SSUBW2**

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register, while the SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 0  | 0  | 1  | 1  | 0  | 1 | 0  | 0  | 1  | 1  | 0  | 0  | 0  | Rm | 0  | 0  | 1  | 1  | 0  | 0  | Rn | Rd |

SSUBW{2} <Vd>,<Ta>, <Vn>,<Ta>, <Vm>,<Tb>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

```
<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```
Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]
bits(2*\text{datasize}) \text{operand1} = \text{V}[n];
bits(\text{datasize}) \text{operand2} = \text{Vpart}[m, \text{part}];
bits(2*\text{datasize}) \text{result};
\text{integer element1};
\text{integer element2};
integer \text{sum};

\text{for e = 0 to elements-1}
\begin{align*}
\text{element1} &= \text{Int(}\text{Elem}(\text{operand1}, \ e, 2*\text{esize}), \text{unsigned}) ;
\text{element2} &= \text{Int(}\text{Elem}(\text{operand2}, \ e, \text{esize}), \text{unsigned}) ;
\text{if sub_op then}
&\quad \text{sum} = \text{element1} - \text{element2} ;
\text{else}
&\quad \text{sum} = \text{element1} + \text{element2} ;
\text{Elem}[	ext{result}, \ e, 2*\text{esize}] = \text{sum<2*\text{esize}-1:0>} ;
\end{align*}
\text{V}[d] = \text{result} ;
\]

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | x | x | 1 | x | size | Rn | Rt
   L                       opcode

One register (opcode == 0111)

ST1 { <Vt>.<T> }, [<Xn|SP>]

Two registers (opcode == 1010)

ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

Three registers (opcode == 0110)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

Four registers (opcode == 0010)

ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

integer t = UINT(Rt);
integer n = UINT(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | x | x | 1 | x | size | Rn | Rt
   L                       opcode
One register, immediate offset (Rm == 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

**Assembler Symbols**

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in “Q”:
For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#8</td>
</tr>
<tr>
<td>1</td>
<td>#16</td>
</tr>
</tbody>
</table>

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#16</td>
</tr>
<tr>
<td>1</td>
<td>#32</td>
</tr>
</tbody>
</table>

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;  // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;  // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;  // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;  // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;  // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;  // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;  // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;  // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

offs = Zeros();
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
      offs = offs + ebytes;
      tt = (tt + 1) MOD 32;

if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 0  | S  | size | Rn | Rm |
| L  | R  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**8-bit (opcode == 000)**

ST1 { `<Vt>`.[B] }[<index>], [ `<Xn|SP>` ]

**16-bit (opcode == 010 && size == x0)**

ST1 { `<Vt>`.[H] }[<index>], [ `<Xn|SP>` ]

**32-bit (opcode == 100 && size == 00)**

ST1 { `<Vt>`.[S] }[<index>], [ `<Xn|SP>` ]

**64-bit (opcode == 100 && S == 0 && size == 01)**

ST1 { `<Vt>`.[D] }[<index>], [ `<Xn|SP>` ]

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | Q  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 0  | S  | size | Rn | Rm |
| L  | R  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Post-index**

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```
8-bit, immediate offset (Rm == 11111 && opcode == 000)

ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1

8-bit, register offset (Rm != 11111 && opcode == 000)

ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == x0)

ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4

32-bit, register offset (Rm != 11111 && opcode == 100 && size == x0)

ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);  // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);  // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);  // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);  // D[0-1]
            scale = 3;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored. Depending on the settings in the CPACR_ELI, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

<p>| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|</p>
<table>
<thead>
<tr>
<th>Q</th>
<th>0 0 1 1 0 0 0 0</th>
<th>0 0 0 0 0 1 0 0</th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

integer t = \text{UInt}(Rt);
integer n = \text{UInt}(Rn);
integer m = \text{integer UNKNOWN};
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

<p>| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|</p>
<table>
<thead>
<tr>
<th>Q</th>
<th>0 0 1 1 0 0 1 0</th>
<th>0 0 0 0 1 0 0</th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

Immediate offset (Rm == 11111)

ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

integer t = \text{UInt}(Rt);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

- `<Vt>` is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vt2>` is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the post-index immediate offset, encoded in “Q”:
<imm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
   when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
   when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
   when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
   when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
   when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
   when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
   when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
   otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
tt integer;
constant integer ebytes = esize DIV 8;
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];
offs = Zeros();
for r = 0 to rpt-1
  for e = 0 to elements-1
    tt = (t + r) MOD 32;
    for s = 0 to selem-1
      rval = V[tt];
      if memop == MemOp_LOAD then
        Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
        V[tt] = rval;
      else // memop == MemOp_STORE
        Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
      offs = offs + ebytes;
      tt = (tt + 1) MOD 32;
if wback then
  if m != 31 then
    offs = X[m];
  if n == 31 then
    SP[] = address + offs;
  else
    X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **No offset** and **Post-index**

### No offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |
| L | R | opcode |

#### 8-bit (opcode == 000)

ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]

#### 16-bit (opcode == 010 && size == x0)

ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]

#### 32-bit (opcode == 100 && size == 00)

ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]

#### 64-bit (opcode == 100 && S == 0 && size == 01)

ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

### Post-index

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | x | x | 0 | S | size | Rn | Rt |
| L | R | opcode |
8-bit, immediate offset (Rm == 11111 && opcode == 000)
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2

8-bit, register offset (Rm != 11111 && opcode == 000)
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
Shared Decode

integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);    // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);    // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);    // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);    // D[0-1]
    scale = 3;
  MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

### No offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>L</td>
</tr>
</tbody>
</table>

opcode

```cpp
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

```cpp
integer t = UINT(Rt);
integer n = UINT(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### Post-index

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>L</td>
</tr>
</tbody>
</table>

opcode

**Immediate offset (Rm == 11111)**

```cpp
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

**Register offset (Rm != 11111)**

```cpp
ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```cpp
integer t = UINT(Rt);
integer n = UINT(Rn);
integer m = UINT(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

### Assembler Symbols

- `<Vt>` Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vt2>` Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
- `<Vt3>` Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt; // number of iterations
integer selem; // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4; // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1; // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3; // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1; // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1; // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

ST3 (multiple structures)
Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

### No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | O  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn | Rt |

L   R   opcode

8-bit (opcode == 001)


16-bit (opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]

32-bit (opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]

64-bit (opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

### Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| O  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn | Rt |

L   R   opcode

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
8-bit, immediate offset (Rm == 11111 && opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3

8-bit, register offset (Rm != 11111 && opcode == 001)

ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
    scale = UInt(size);
    replicate = TRUE;
  when 0
    index = UInt(Q:S:size);  // B[0-15]
  when 1
    if size<0> == '1' then UNDEFINED;
    index = UInt(Q:S:size<1>);  // H[0-7]
  when 2
    if size<1> == '1' then UNDEFINED;
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
    else
      if S == '1' then UNDEFINED;
      index = UInt(Q);  // D[0-1]
    scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| L  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opcode |


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| L  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| opcode |

Immediate offset (Rm == 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], < Xm>

integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#32</td>
</tr>
<tr>
<td>1</td>
<td>#64</td>
</tr>
</tbody>
</table>

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

**Shared Decode**

```plaintext
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
  when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
  when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
  when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
  when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
  when '0111' rpt = 3; selem = 1;    // LD/ST1 (1 register)
  when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
  otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

ST4 (multiple structures)
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
                offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ST4 (single structure)**

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

### No offset

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>001</td>
<td>00</td>
</tr>
<tr>
<td>16-bit</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>32-bit</td>
<td>101</td>
<td>00</td>
</tr>
<tr>
<td>64-bit</td>
<td>101</td>
<td>00</td>
</tr>
</tbody>
</table>

#### 8-bit (opcode == 001)


#### 16-bit (opcode == 011 && size == x0)


#### 32-bit (opcode == 101 && size == 00)


#### 64-bit (opcode == 101 && S == 0 && size == 01)


```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

### Post-index

<table>
<thead>
<tr>
<th>opcode</th>
<th>Rm</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>001</td>
<td>00</td>
</tr>
<tr>
<td>16-bit</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>32-bit</td>
<td>101</td>
<td>00</td>
</tr>
</tbody>
</table>

#### Post-index

```plaintext
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```
8-bit, immediate offset (Rm == 11111 & opcode == 001)

8-bit, register offset (Rm != 11111 & opcode == 001)

16-bit, immediate offset (Rm == 11111 & opcode == 011 & size == x0)

16-bit, register offset (Rm != 11111 & opcode == 011 & size == x0)

32-bit, immediate offset (Rm == 11111 & opcode == 101 & size == 00)

32-bit, register offset (Rm != 11111 & opcode == 101 & size == 00)
ST4 \{ <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S \}[<index>], [<Xn|SP>], <Xm>

64-bit, immediate offset (Rm == 11111 & opcode == 101 & S == 0 & size == 01)
ST4 \{ <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D \}[<index>], [<Xn|SP>], #32

64-bit, register offset (Rm != 11111 & opcode == 101 & S == 0 & size == 01)
ST4 \{ <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D \}[<index>], [<Xn|SP>], <Xm>

integer t = UInt(Rt);
integer n = Uint(Rn);
integer m = Uint(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".
For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".
For the 32-bit variant: is the element index, encoded in "Q:S".
For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.
integer init_scale = Uint(opcode<2:1>);
integer scale = init_scale;
integer selem = Uint(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
   when 3  
      // load and replicate
      if L == '0' || S == '1' then UNDEFINED;
      scale = Uint(size);
      replicate = TRUE;
   when 0
      index = Uint(Q:S:size);        // B[0-15]
   when 1
      if size<0> == '1' then UNDEFINED;
      index = Uint(Q:S:size<1>);    // H[0-7]
   when 2
      if size<1> == '1' then UNDEFINED;
      if size<0> == '0' then
         index = Uint(Q:S);         // S[0-3]
      else
         if S == '1' then UNDEFINED;
         index = Uint(Q);          // D[0-1]
         scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
Operation

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
            offs = offs + ebytes;
            t = (t + 1) MOD 32;
if wback then
    if m != 31 then
        offs = X[m];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n] = address + offs;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see Load/Store SIMD and Floating-point Non-temporal pair.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>opc</th>
<th>1 0 1 1 0 0 0 0</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

32-bit (opc == 00)

STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]

64-bit (opc == 01)

STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]

128-bit (opc == 10)

STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]

// Empty.

**Assembler Symbols**

- `<Dt1>`  Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Dt2>`  Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<Qt1>`  Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<Qt2>`  Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<St1>`  Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
- `<St2>`  Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
- `<Xn|SP>`  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`  For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as `<imm>/4`.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as `<imm>/8`.

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as `<imm>/16`.

**Shared Decode**

```javascript
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
```
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VECSTREAM] = data1;
Mem[address+dbytes, dbytes, AccType_VECSTREAM] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Signed offset

Post-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| opc | 1   | 0   | 1   | 0   | 1   | 0   | imm7|     | Rt2 |     | Rn  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| opc | 1   | 0   | 1   | 1   | 0   | 0   |imm7|     | Rt2 |     | Rn  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>], #<imm>!

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>!

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>!

boolean wback = TRUE;
boolean postindex = FALSE;

Signed offset

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| opc | 1   | 0   | 1   | 1   | 0   | 1   | 0   |imm7|     | Rt2 |     | Rn  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

boolean wback = TRUE;
boolean postindex = TRUE;
32-bit (opc == 00)

STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]

64-bit (opc == 01)

STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]

128-bit (opc == 10)

STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]

boolean wback = FALSE;
boolean postindex = FALSE;

Assembler Symbols

<Dt1>  Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>  Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>  Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>  Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>  Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>  Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.
For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.
For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.
For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.
For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.
For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bpiration offset = bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

if !postindex then
    address = address + offset;

data1 = V[t];
data2 = V[t2];
Mem[address, dbytes, AccType_VEC] = data1;
Mem[address+dbytes, dbytes, AccType_VEC] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index, Pre-index and Unsigned offset

**Post-index**

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>0</th>
<th>0</th>
<th>imm9</th>
<th>0</th>
<th>1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 00)

STR <Bt>, [<Xn|SP>], #<simm>

16-bit (size == 01 && opc == 00)

STR <Ht>, [<Xn|SP>], #<simm>

32-bit (size == 10 && opc == 00)

STR <St>, [<Xn|SP>], #<simm>

64-bit (size == 11 && opc == 00)

STR <Dt>, [<Xn|SP>], #<simm>

128-bit (size == 00 && opc == 10)

STR <Qt>, [<Xn|SP>], #<simm>

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

**Pre-index**

| size | 1 | 1 | 1 | 1 | 0 | 0 | x | 0 | 0 | imm9 | 1 | 1 | Rn | Rt |
|------|---|---|---|---|---|---|---|---|------|---|---|----|----|
| opc  |   |   |   |   |   |   |   |   |      |   |   |    |    |
8-bit (size == 00 && opc == 00)

\[
\text{STR } <\text{Bt}>, [\text{<Xn|SP>}, \#<\text{simm}>]!
\]

16-bit (size == 01 && opc == 00)

\[
\text{STR } <\text{Ht}>, [\text{<Xn|SP>}, \#<\text{simm}>]!
\]

32-bit (size == 10 && opc == 00)

\[
\text{STR } <\text{St}>, [\text{<Xn|SP>}, \#<\text{simm}>]!
\]

64-bit (size == 11 && opc == 00)

\[
\text{STR } <\text{Dt}>, [\text{<Xn|SP>}, \#<\text{simm}>]!
\]

128-bit (size == 00 && opc == 10)

\[
\text{STR } <\text{Qt}>, [\text{<Xn|SP>}, \#<\text{simm}>]!
\]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Signed offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 1 | 1 | 1 | 1 | 0 | 1 | x | 0 | imm12 | Rn | Rt |
| opc |

8-bit (size == 00 && opc == 00)

\[
\text{STR } <\text{Bt}>, [\text{<Xn|SP>}{, \#<\text{pimm}>}]
\]

16-bit (size == 01 && opc == 00)

\[
\text{STR } <\text{Ht}>, [\text{<Xn|SP>}{, \#<\text{pimm}>}]
\]

32-bit (size == 10 && opc == 00)

\[
\text{STR } <\text{St}>, [\text{<Xn|SP>}{, \#<\text{pimm}>}]
\]

64-bit (size == 11 && opc == 00)

\[
\text{STR } <\text{Dt}>, [\text{<Xn|SP>}{, \#<\text{pimm}>}]
\]

128-bit (size == 00 && opc == 10)

\[
\text{STR } <\text{Qt}>, [\text{<Xn|SP>}{, \#<\text{pimm}>}]
\]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<It> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
if HaveMTEExt() then
  SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

if !postindex then
  address = address + offset;

case memop of
  when MemOp_STORE
    data = V[t];
    Mem[address, datasize DIV 8, AccType_VEC] = data;
  when MemOp_LOAD
    data = Mem[address, datasize DIV 8, AccType_VEC];
    V[t] = data;

if wback then
  if postindex then
    address = address + offset;
  if n == 31 then
    SP[] = address;
  else
    X[n] = address;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1 1 1 0 0 x 0 1</th>
<th>Rm</th>
<th>option</th>
<th>S</th>
<th>1 0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

8-fsreg,STR-8-fsreg (size == 00 && opc == 00 && option != 011)

STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>)}

8-fsreg,STR-8-fsreg (size == 00 && opc == 00 && option == 011)

STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}}

16-fsreg,STR-16-fsreg (size == 01 && opc == 00)

STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

32-fsreg,STR-32-fsreg (size == 10 && opc == 00)

STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

64-fsreg,STR-64-fsreg (size == 11 && opc == 00)

STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

128-fsreg,STR-128-fsreg (size == 00 && opc == 10)

STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED; // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend> For the 8-bit variant: is the index extend specifier, encoded in "option":

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in "option":

<table>
<thead>
<tr>
<th>option</th>
<th>&lt;extend&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>UXTW</td>
</tr>
<tr>
<td>110</td>
<td>SXTW</td>
</tr>
<tr>
<td>111</td>
<td>SXTX</td>
</tr>
</tbody>
</table>

For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.
For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#1</td>
</tr>
</tbody>
</table>

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#2</td>
</tr>
</tbody>
</table>

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#3</td>
</tr>
</tbody>
</table>

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

<table>
<thead>
<tr>
<th>S</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0</td>
</tr>
<tr>
<td>1</td>
<td>#4</td>
</tr>
</tbody>
</table>

**Shared Decode**

```plaintext
integer n =UInt(Rn);
integer t =UInt(Rt);
integer m =UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;
```
Operation

bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>size</th>
<th>1 1 1 1</th>
<th>0 0</th>
<th>x 0 0</th>
<th>imm9</th>
<th>0 0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8-bit (size == 00 && opc == 00)

STUR <Bt>, [<Xn|SP>{, #<simm>}]

16-bit (size == 01 && opc == 00)

STUR <Ht>, [<Xn|SP>{, #<simm>}]

32-bit (size == 10 && opc == 00)

STUR <St>, [<Xn|SP>{, #<simm>}]

64-bit (size == 11 && opc == 00)

STUR <Dt>, [<Xn|SP>{, #<simm>}]

128-bit (size == 00 && opc == 10)

STUR <Qt>, [<Xn|SP>{, #<simm>}]

integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

integer n = UInt(Rn);
integer t = UInt(Rt);

MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
Operation

```c
if HaveMTEExt() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t];
        Mem[address, datasize DIV 8, AccType_VEC] = data;
    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.
**SUB (vector)**

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | Rm | Rn | Rd | U  |
```

**Scalar**<V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

**Vector**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | Rn | Rd | U  |
```

**Vector**<Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

**Assembler Symbols**

| <V> | Is a width specifier, encoded in “size”:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<p>| &lt;d&gt; | Is the number of the SIMD&amp;FP destination register, in the &quot;Rd&quot; field. |
|     | &lt;n&gt; | Is the number of the first SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field. |
|     | &lt;m&gt; | Is the number of the second SIMD&amp;FP source register, encoded in the &quot;Rm&quot; field. |
|     | &lt;Vd&gt; | Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field. |
|     | &lt;T&gt; | Is an arrangement specifier, encoded in &quot;size:Q&quot;. |</p>
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SUBHN, SUBHN2

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see RSUBHN.

The SUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|   0 |   Q |   0 |   0 |   1 |   1 |   1 |   0 |   size |   1 |   Rm |   0 |   1 |   1 |   0 |   0 |   0 |   Rn |   Rd |
```

SUBHN(2) <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```java
def integer d = UInt(Rd);
def integer n = UInt(Rn);
def integer m = UInt(Rm);

if size == '11' then UNDEFINED;
def integer esize = 8 << UInt(size);
def integer datasize = 64;
def integer part = UInt(Q);
def integer elements = datasize DIV esize;

bool sub_op = (o1 == '1');
def boolean round = (U == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(2*datasize) operand2 = V[m];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;
Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUDOT (by element)**

Dot product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2, this is an **OPTIONAL** instruction. **ID_AA64ISAR0_EL1.** I8MM indicates whether this instruction is supported.

### Vector (Armv8.6)

![Vector Table]

- **US**
- **<Vd>** Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- **<Ta>** Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Tb>** Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
- **<index>** Is the immediate index of a quadtuplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.

---

*SUDOT (by element) Page 1310*
Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d] = result;
\end{verbatim}
SUQADD

Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

SUQADD <V><<d>, <V><<n>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector

SUQADD <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "size:Q":
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SXTL, SXTL2

Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values. The SXTL instruction extracts the source vector from the lower half of the source register, while the SXTL2 instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of **SSHLL, SSHLL2**. This means:

- The encodings in this description are named to match the encodings of **SSHLL, SSHLL2**.
- The description of **SSHLL, SSHLL2** gives the operational pseudocode for this instruction.

![Binary representation](image)

SXTL{2} <Vd>, <Ta>, <Vn>, <Tb>

is equivalent to

SSHLL{2} <Vd>, <Ta>, <Vn>, <Tb>, #0

and is the preferred disassembly when `BitCount(immh) == 1`.

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;Vd&gt;</th>
<th>Is the name of the SIMD&amp;FP destination register, encoded in the &quot;Rd&quot; field.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&lt;Ta&gt;</th>
<th>Is an arrangement specifier, encoded in “immh”:</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td><strong>SEE Advanced SIMD modified immediate</strong></td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>&lt;Vn&gt;</th>
<th>Is the name of the SIMD&amp;FP source register, encoded in the &quot;Rn&quot; field.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&lt;Tb&gt;</th>
<th>Is an arrangement specifier, encoded in “immh:Q”:</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
</tr>
</tbody>
</table>

**Operation**

The description of **SSHLL, SSHLL2** gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Two register table (len == 01)

TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>

Three register table (len == 10)

TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>

Four register table (len == 11)

TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>

Single register table (len == 00)

TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');

Assembler Symbols

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "Q":

For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
bits(128*regs) table = Zeros();
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Two register table (len == 01)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>

Three register table (len == 10)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>

Four register table (len == 11)

TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>

Single register table (len == 00)

TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.
<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.
<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.
<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m];
b(R128*regs) table = Zeros();
b(Rdatasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n];
    n = (n + 1) MOD 32;
result = if is_tbl then Zeros() else V[d];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**TRN1**

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

By using this instruction with TRN2, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.

```
0 1 2 3
Vn

3 2 1 0
Vd

3 2 1 0
Vm
```

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 0 0 1 1 1 0 size 0 Rm 0 0 1 0 1 0 Rn Rd
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

```
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
```

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

By using this instruction with TRN1, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.

depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

| <Vd> | Is the name of the SIMD&FP destination register, encoded in the "Rd" field. |
| <T>  | Is an arrangement specifier, encoded in "size:Q": |
| size | Q  | <T> |
| 00   | 0  | 8B |
| 00   | 1  | 16B|
| 01   | 0  | 4H |
| 01   | 1  | 8H |
| 10   | 0  | 2S |
| 10   | 1  | 4S |
| 11   | 0  | RESERVED |
| 11   | 1  | 2D |

| <Vm> | Is the name of the second SIMD&FP source register, encoded in the "Rm" field. |
| <Vn> | Is the name of the first SIMD&FP source register, encoded in the "Rn" field. |
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 Q 1 0 1 1 1 0 size 1 Rm 0 1 1 1 1 1 Rn Rd
```

UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register, while the UABAL2 instruction extracts each source vector from the upper half of each source register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

The assembler code for the UABAL instruction is:

```
UABAL2 <Vd>, <Ta>, <Vn>, <Vm>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

\[
\begin{array}{ccc}
\text{size} & \text{Q} & \text{<T>} \\
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & \text{RESERVED}
\end{array}
\]

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
element1 = Int(Elem[operand1, e, esize], unsigned);
element2 = Int(Elem[operand2, e, esize], unsigned);
absdiff = Abs(element1-element2)<esize-1:0>;
Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UABDL, UABDL2**

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The **UABDL** instruction extracts each source vector from the lower half of each source register, while the **UABDL2** instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 | Q | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 0 | 1 | 1 | 0 | 0 | Rn | Rd |
   | U | op |
```

**UABDL**<sub>2</sub> 2 <Vd>.<Ta>, <Vn>.<Tb>

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

**Assembler Symbols**

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
  0 | [absent]
  1 | [present]
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

```
  size  <Ta>
  00  8H
  01  4S
  10  2D
  11  RESERVED
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

```
  size  Q  <Tb>
  00  0  8B
  00  1  16B
  01  0  4H
  01  1  8H
  10  0  2S
  10  1  4S
  11  x  RESERVED
```

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UADALP**

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

**Assembler Symbols**

- `<Vd>` is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<Ta>` is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` is the name of the SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bias(datasize) operand = V[n];
bias(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UADDL instruction extracts each source vector from the lower half of each source register, while the UADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
UADDL{2} <Vd>, <Ta>, <Vn>, <Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>Rm</th>
<th></th>
<th></th>
<th>Rn</th>
<th></th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

UADDL2 <Vd>, <Ta>, <Vn>, <Tb>, <Vm>, <Tb>

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":";

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>Q</th>
<th></th>
<th></th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;Ta&gt;</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;Ta&gt;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;Ta&gt;</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;Ta&gt;</td>
<td></td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\[
\begin{array}{cccccccccccccccc}
\hline
0 & Q & 1 & 0 & 1 & 1 & 0 & \text{size} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & Rn & Rd & \hline
\end{array}
\]

UADDLP `<Vd>.<Ta>`, `<Vn>.<Tb>`

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);

\[
\begin{align*}
\text{if size} & \text{ == '11' then UNDEFINED;} \\
\text{integer esize} & \text{ = 8 \text{ << UInt(size);} \\
\text{integer dataisize} & \text{ = if Q == '1' then 128 else 64;} \\
\text{integer elements} & \text{ = dataisize DIV (2 \ast esize);} \\
\text{boolean acc} & \text{ = (op == '1);} \\
\text{boolean unsigned} & \text{ = (U == '1');}
\end{align*}
\]

Assembler Symbols

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{Ta}>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1D</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<\text{Vn}>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Tb}>\) Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register
together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as
the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and
Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

- `<V>` is the destination width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>` is the number of the SIMD&FP destination register, encoded in the “Rd” field.
- `<Vn>` is the name of the SIMD&FP source register, encoded in the “Rn” field.
- `<T>` is an arrangement specifier, encoded in “size:Q”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);
V[d] = sum<2*esize-1:0>;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.
UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register, while the UADDW2 instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

|   | Q | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 0 | 1 | 0 | 0 | Rn | Rd |   |
| U |   |   |   |   |   |   |     |   |   |   |   |   |   |   |   |   |   |   |   |

UADDW[2] <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
<td></td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, 2*esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UCVTF (vector, fixed-point)**

Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the **FPCR**, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see **Floating-point exception traps**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 1 1 1 1 1 1</td>
</tr>
<tr>
<td>integer d = UInt(Rd);</td>
</tr>
<tr>
<td>integer n = UInt(Rn);</td>
</tr>
<tr>
<td>if immh == '000x'</td>
</tr>
<tr>
<td>integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;</td>
</tr>
<tr>
<td>integer datasize = esize;</td>
</tr>
<tr>
<td>integer elements = 1;</td>
</tr>
<tr>
<td>integer fracbits = (esize * 2) - UInt(immh:immb);</td>
</tr>
<tr>
<td>boolean unsigned = (U == '1');</td>
</tr>
<tr>
<td>FPRounding rounding = FPRoundingMode(FPCR[]);</td>
</tr>
</tbody>
</table>

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 1 0 1 1 1</td>
</tr>
<tr>
<td>integer d = UInt(Rd);</td>
</tr>
<tr>
<td>integer n = UInt(Rn);</td>
</tr>
<tr>
<td>if immh == '0000' then SEE(asimdimm);</td>
</tr>
<tr>
<td>if immh == '000x'</td>
</tr>
<tr>
<td>if immh&lt;3&gt;:Q == '10' then UNDEFINED;</td>
</tr>
<tr>
<td>integer esize = if immh == '1xxx' then 64 else if immh == '01xx' then 32 else 16;</td>
</tr>
<tr>
<td>integer datasize = if Q == '1' then 128 else 64;</td>
</tr>
<tr>
<td>integer elements = datasize DIV esize;</td>
</tr>
<tr>
<td>integer fracbits = (esize * 2) - UInt(immh:immb);</td>
</tr>
<tr>
<td>boolean unsigned = (U == '1');</td>
</tr>
<tr>
<td>FPRounding rounding = FPRoundingMode(FPCR[]);</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

**<V>** is a width specifier, encoded in “immh”:
Imm & FP

<table>
<thead>
<tr>
<th>imm</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_AAdvanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>imm</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>imm</th>
<th>&lt;fbits&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_AAdvanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>001x</td>
<td>(32-Uint(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

*CheckFPAdvSIMDEnabled64*();

bits(datasize) operand = V[n];

bits(esize) element;

FPCRT ype fpcr = FPCR[];

boolean merge = elements == 1 && IsMerging(fpcr);

bits(128) result = if merge then V[d] else Zeros();

for e = 0 to elements-1

  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding);

V[d] = result;
**UCVTF (vector, integer)**

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision, Scalar single-precision and double-precision, Vector half precision and Vector single-precision and double-precision

### Scalar half precision

**(Armv8.2)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rd |

UCVTF <Hd>, <Hn>

```plaintext
if !HaveFP16Ext() then UNDEFINED;
```

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

### Scalar single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | Rd |

UCVTF <V<d>, <V<n>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
```

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

### Vector half precision

**(Armv8.2)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | Rd |

UCVTF (vector, integer)
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Vector single-precision and double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | | | | | | | | | | |
| U  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | | | | | | | | | | |

UCVTF <Vd>, <T>, <Vn>, <T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<V> Is a width specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.
<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the half-precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q”:

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>RESERVE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d] else Zeros();

FPRounding rounding = FPRoundingMode(fpcr);
better(elementsize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding);
V[d] = result;
```
**UCVTF (scalar, fixed-point)**

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the FPCR, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in FPCR, the exception results in either a flag being set in FPSR, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ftype</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>scale</td>
<td></td>
<td>Rn</td>
<td></td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

rmode opcode
32-bit to half-precision (sf == 0 & ftype == 11) (Armv8.2)

UCVTF <Hd>, <Wn>, #<fbits>

32-bit to single-precision (sf == 0 & ftype == 00)

UCVTF <Sd>, <Wn>, #<fbits>

32-bit to double-precision (sf == 0 & ftype == 01)

UCVTF <Dd>, <Wn>, #<fbits>

64-bit to half-precision (sf == 1 & ftype == 11) (Armv8.2)

UCVTF <Hd>, <Xn>, #<fbits>

64-bit to single-precision (sf == 1 & ftype == 00)

UCVTF <Sd>, <Xn>, #<fbits>

64-bit to double-precision (sf == 1 & ftype == 01)

UCVTF <Dd>, <Xn>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;  
FPRounding rounding;

case ftype of
  when '00' fltsize = 32;
  when '01' fltsize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;
  end;
if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits> For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale".
For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

**Operation**

```c
CheckFPAdvSIMDEnabled64();

FPCRType fpcr = FPCR[4];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;
intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem(fltval, 0, fltsize) = FixedToFP(intval, fracbits, TRUE, fpcr, rounding);
V[d] = fltval;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UCVTF (scalar, integer)**

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the **FPCR**, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in **FPCR**, the exception results in either a flag being set in **FPSR**, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| sf  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
```

*sf*: Source format (0 for integer, 1 for floating-point)

*ftype*: Floating-point type

*rmode*: Rounding mode

*opcode*: Instruction opcode
32-bit to half-precision (sf == 0 && ftype == 11) (Armv8.2)
UCVTF <Hd>, <Wn>

32-bit to single-precision (sf == 0 && ftype == 00)
UCVTF <Sd>, <Wn>

32-bit to double-precision (sf == 0 && ftype == 01)
UCVTF <Dd>, <Wn>

64-bit to half-precision (sf == 1 && ftype == 11) (Armv8.2)
UCVTF <Hd>, <Xn>

64-bit to single-precision (sf == 1 && ftype == 00)
UCVTF <Sd>, <Xn>

64-bit to double-precision (sf == 1 && ftype == 01)
UCVTF <Dd>, <Xn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
  when '00'
    fltsize = 32;
  when '01'
    fltsize = 64;
  when '10'
    UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      fltsize = 16;
    else
      UNDEFINED;

rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn> Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn> Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
Operation

CheckFPAdvSIMDEnabled64();

FPCRTypedefpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = if merge then V[d] else Zeros();
Elem(fltval, 0, fltsize) = FixedToFP(intval, 0, TRUE, fpcr, rounding);
V[d] = fltval;
UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it. ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector
(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | size | L  | M  | Rm | 1  | 1  | 1  | 0  | H  | 0  | Rn | Rd |

UDOT <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.4B[<index>]

if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d =UInt(Rd);
integer n =UInt(Rn);
integer m =UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Ta> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>0</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Tb> Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>0</th>
<th>8B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.
#index> Is the element index, encoded in the "H:L" fields.
Operation

`CheckFPAdvSIMDEnabled64();`

```c
bits(datasize) operand1 = V[n];
bits(128) operand2 = V[m];
bits(datasize) result = V[d];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        
        res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
```
UDOT (vector)

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

`ID_AAA41SAR0_EL1`. DP indicates whether this instruction is supported.

### Vector

(Armv8.2)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| ![Vector Diagram](UDOT_vector.png) |

UDOT `<Vd>.,<Ta>.,<Vn>.,<Tb>.,<Vm>.,<Tb>`

if ![HaveDOTPext()](UDOT_HaveDOTPext.png) then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = `UInt`(Rd);
integer n = `UInt`(Rn);
integer m = `UInt`(Rm);
integer esize = 8 << `UInt`(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

### Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- `<Ta>` Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;Ta&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Tb>` Is an arrangement specifier, encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

\textbf{CheckFPAdvSIMDEnabled64(\textit{\)}};

\begin{verbatim}
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

result = V[d];
for \(e = 0\) to \(\text{elements-1}\)
    integer res = 0;
    integer element1, element2;
    for \(i = 0\) to \(3\)
        if signed then
            element1 = \textit{Sint}(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = \textit{Sint}(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = \textit{UInt}(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = \textit{UInt}(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d] = result;
\end{verbatim}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UHADD

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see **URHADD**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rn</td>
<td>1</td>
<td>Rd</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UHADD <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size == '11' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- boolean unsigned = (U == '1');

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    Elem[result, e, esize] = sum<esize:1>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];

bits(datasize) operand2 = V[m];

bits(datasize) result;

integer element1;

integer element2;

integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    Elem[result, e, esize] = diff<esize:1>;

V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in "size.Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
  element1 = Int(Elem(operand1, e, esize), unsigned);
  element2 = Int(Elem(operand2, e, esize), unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem(result, e, esize) = maxmin<esize-1:0>;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UMAXP**

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th></th>
<th>O</th>
<th>Q</th>
<th>size</th>
<th>Rm</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>o1</td>
</tr>
</tbody>
</table>

UMAXP `<Vd>..<T>`, `<Vn>..<T>`, `<Vm>..<T>`

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```java
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
  element1 = Int(Elem[concat, 2*e, esize], unsigned);
  element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
  maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
  Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UMAXV**

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Image of binary representation](image)

**Assembler Symbols**

- `<V>`: Is the destination width width specifier, encoded in "size":

<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

- `<Vn>`: Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- `<T>`: Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>Q</code></th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements - 1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d] = maxmin<esize-1:0>;
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Assembler Symbols

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem(operand1, e, esize), unsigned);
    element2 = Int(Elem(operand2, e, esize), unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem(result, e, esize) = maxmin<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

UMIN
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UMINP**

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|----------------|
| **0** | **0** | **1** | **0** | **1** | **1** | **0** | **Rm** | **1** | **0** | **1** | **1** | **Rn** | **Rd** |
| **U** | **01** |

**UMINP** `<Vd>.<T>, <Vn>.<T>, <Vm>.<T>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

**Assembler Symbols**

- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Vn>`: Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>`: Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;
for e = 0 to elements-1
    element1 = Int(Elem(concat, 2*e, esize], unsigned);
    element2 = Int(Elem(concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;
V[d] = result;
```

**UMINP**
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMINV**

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

**Asmblr Symbols**

- `<V>` is the destination width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<d>` is the number of the SIMD&FP destination register, encoded in the “Rd” field.

- `<Vn>` is the name of the SIMD&FP source register, encoded in the “Rn” field.

- `<T>` is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

```plaintext
if unsigned then
    maxmin = Int(Elem[operand, 0, esize], unsigned);
else
    maxmin = Int(Elem[operand, 0, esize], unsigned);
end if
```

```plaintext
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
end for
```

```plaintext
V[d] = maxmin<esize-1:0>;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
### UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

#### Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
<Vm> Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

@index> Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register, while the UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | Q | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 0 | 0 | 0 | Rn | Rd | 01 |

UMLAL\{2\} <Vd>, <Ta>, <Vn>, <Tb>

integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
datasize = 64;
part = UInt(Q);
integer elements = datasize DIV esize;
sub_op = (o1 == '1');
unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>size</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  product = (element1*element2)<2*esize-1:0>;
  if sub_op then
    accum = Elem[operand3, e, 2*esize] - product;
  else
    accum = Elem[operand3, e, 2*esize] + product;
  Elem[result, e, 2*esize] = accum;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The **UMLSL** instruction extracts vector elements from the lower half of the first source register, while the **UMLSL2** instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
  otherwise UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
<th>Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ta>** Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>45</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Tb>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>0:Rm</td>
</tr>
<tr>
<td>10</td>
<td>M:Rm</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Restricted to V0-V15 when element size <Ts> is H.

Is an element size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the element index, encoded in "size:L:H:M":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
# UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMLSL instruction extracts each source vector from the lower half of each source register, while the UMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```plaintext
UMLSL{2} <Vd>,<Ta>, <Vn>,<Tb>, <Vm>.<Tb>
```

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

### Assembler Symbols

**2** Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Ta>** Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

**<Tb>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) operand3 = V[d];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an \textit{OPTIONAL} instruction. \textit{ID AA64ISAR0 EL1}.I8MM indicates whether this instruction is supported.

**Vector**

\textbf{(Armv8.6)}

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U| 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rm| 1  | 0  | 1  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rd| 1  | 0  | 1  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

\texttt{UMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B}

\[ \text{if } !\text{HaveInt8MatMulExt}() \text{ then UNDEFINED; } \]
\[ \text{integer } n = \text{UInt}(Rn); \]
\[ \text{integer } m = \text{UInt}(Rm); \]
\[ \text{integer } d = \text{UInt}(Rd); \]

**Assembler Symbols**

\(<Vd>\) Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

\(<Vn>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<Vm>\) Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

\texttt{CheckFPAdvSIMDEnabled64();}

\texttt{bits(128) operand1 = V[n];}

\texttt{bits(128) operand2 = V[m];}

\texttt{bits(128) addend = V[d];}

\texttt{V[d] = MatMulAdd(addend, operand1, operand2, TRUE, TRUE);}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias `MOV (to general)`.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | imm5   | 0 | 0 | 1 | 1 | 1 | Rn | Rd |

32-bit (Q == 0)

UMOV <Wd>, <Vn>.<Ts>[<index>]

64-reg,UMOV-64-reg (Q == 1 && imm5 == x1000)

UMOV <Xd>, <Vn>.<Ts>[<index>]

integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
  when '0xxxx1' size = 0;    // UMOV Wd, Vn.B
  when '0xxx10' size = 1;    // UMOV Wd, Vn.H
  when '0xx100' size = 2;    // UMOV Wd, Vn.S
  when '1x1000' size = 3;    // UMOV Xd, Vn.D
  otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
</tbody>
</table>

For the 64-reg,UMOV-64-reg variant: is an element size specifier, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>&lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxx10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xx100</td>
<td>RESERVED</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
</tbody>
</table>

<index> For the 32-bit variant: is the element index encoded in “imm5”: 
For the 64-reg, UMOV-64-reg variant: is the element index encoded in "imm5<4>".

### Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (to general)</td>
<td>imm5 == 'x100'</td>
</tr>
<tr>
<td>MOV (to general)</td>
<td>imm5 == 'xx100'</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsiz) operand = V[n];
X[d] = ZeroExtend(Elem[operand, index, esize], datasize);
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMULL instruction extracts vector elements from the lower half of the first source register, while the UMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\[
\begin{array}{cccccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 1 & \text{size} & L & M & \text{Rm} & 1 & 0 & 1 & 0 & H & 0 & \text{Rn} & \text{Rd} & U
\end{array}
\]

**Assembler Symbols**

2 \hspace{1cm} Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> \hspace{1cm} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> \hspace{1cm} Is an arrangement specifier, encoded in "size":

\[
\begin{array}{cccc}
00 & \text{RESERVED} \\
01 & 4S \\
10 & 2D \\
11 & \text{RESERVED}
\end{array}
\]

<Vn> \hspace{1cm} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> \hspace{1cm} Is an arrangement specifier, encoded in "size:Q":

\[
\begin{array}{cccc}
00 & x & \text{RESERVED} \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & \text{RESERVED}
\end{array}
\]

<Vm> \hspace{1cm} Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

integer idxds = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = UInt(H:L:M); Rmhi = '0';
  when '10' index = UInt(H:L); Rmhi = M;
otherwise UNDEFINED;

integer d = UInt(Rd);
iinteger n = UInt(Rn);
iinteger m = UInt(Rmhi:Rm);
integer esize = 8 << UInt(size);
iinteger datadsize = 64;
iinteger part = UInt(Q);
iinteger elements = datasize DIV esize;
boolean unsigned = (U == '1');
Restricted to V0-V15 when element size `<Ts>` is H.

Is an element size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;Ts&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Is the element index, encoded in "size:L:H:M”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;index&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H:L:M</td>
</tr>
<tr>
<td>10</td>
<td>H:L</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(idxdsize) operand2 = V[m];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UMULL, UMULL2 (vector)**

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register, while the UMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Register Format](image)

**UMULL{2} <Vd>.,<Ta>, <Vn>.<Tb>, <Vm>.<Tb>**

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

**Assembler Symbols**

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd>  Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<Ta>  Is an arrangement specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn>  Is the name of the first SIMD&FP source register, encoded in the ”Rn” field.

<Tb>  Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  Elem[result, e, 2*esize] = (element1*element2)<<2*esize-1:0;
```

V[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQADD

Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register. If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0  1  1  1  1  1  0  | size | 1          | Rm | 0  0  0  0  1  1          | Rn | Rd |
U

UQADD <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 « UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');

Vector

31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0  1  0  1  1  1  0  | size | 1          | Rm | 0  0  0  0  1  1          | Rn | Rd |
U

UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 « UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q";
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQRSHL

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see UQSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rs</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 0</td>
<td>1</td>
<td>Rm</td>
<td>0 1 0 1 1 1</td>
</tr>
</tbody>
</table>

UQRSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

**Vector**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rs</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 0</td>
<td>1</td>
<td>Rm</td>
<td>0 1 0 1 1 1</td>
</tr>
</tbody>
</table>

UQRSHL <Vd><T>, <Vn><T>, <Vm><T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

**Assembler Symbols**

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQRSHRN, UQRSHRN2

Unsigned saturating Rouded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see UQRSHRN.

The UQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U   | immh | op |

UQRSHRN <Vb><d>, <Va><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U   | immh | op |

UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SRE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```
Assembler Symbols

2  Is the second and upper half specifier. If present it causes the operation to be performed on the upper
64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.
<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.
<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
<Va> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the “Rn” field.
<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits,
encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits,
encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

CheckFPAdvSIMDEnabled64();
bv2(bits(datasize*2) operand = V[n]);
bv2(bits(datasize) result);
integer round_const = if round then (1l << (shift - 1)) else 0;
integer element;
boolean sat;
for e = 0 to elements-1
  element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
  (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
  if sat then FPSR.QC = '1';

Vpart[d, part] = result;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSHL (immediate)

Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector.

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>!= 0000</th>
<th>immh</th>
<th>0 1 1 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>immh</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UQSHL <V><d>, <V><n>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>!= 0000</th>
<th>immh</th>
<th>0 1 1 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>immh</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
  when '00' UNDEFINED;
  when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
  when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
  when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>2B</td>
</tr>
<tr>
<td>001x</td>
<td>4H</td>
</tr>
<tr>
<td>01xx</td>
<td>8H</td>
</tr>
<tr>
<td>1xxx</td>
<td>2S</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
</tbody>
</table>

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>2B</td>
</tr>
<tr>
<td>001x</td>
<td>4H</td>
</tr>
<tr>
<td>01xx</td>
<td>8H</td>
</tr>
<tr>
<td>1xxx</td>
<td>2S</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
</tbody>
</table>

<T> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
</tr>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>2B</td>
</tr>
<tr>
<td>001x</td>
<td>4H</td>
</tr>
<tr>
<td>01xx</td>
<td>8H</td>
</tr>
<tr>
<td>1xxx</td>
<td>2S</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(UInt(immh:immb)-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(UInt(immh:immb)-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(UInt(immh:immb)-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(UInt(immh:immb)-64)</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand = V[n];
bits(datasize) result;
integer element;
boolean sat;
for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see UQRSHL.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 1 1 1 1 1 0 | size | 1 | Rm | 0 1 0 | 0 1 1 | Rn | Rd |

UQSHL <V><d>, <V><n>, <V><m>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 1 1 1 1 0 | size | 1 | Rm | 0 1 0 | 0 1 1 | Rn | Rd |

UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem[operand2, e, esize]<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
        if saturating then
            (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
            if sat then FPSR.QC = '1';
        else
            Elem[result, e, esize] = element<esize-1:0>;
    end if
end for

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see UQRSHRN.

The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 1 1 1 1 1 0 != 0000  immh 1 0 0 1 0 1 Rn Rd

UQSHRN <Vb><d>, <Va><n>, #<shift>
```

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- if immh == '0000' then UNDEFINED;
- if immh<3> == '1' then UNDEFINED;
- integer esize = 8 << HighestSetBit(immh);
- integer datasize = esize;
- integer elements = 1;
- integer part = 0;
- integer shift = (2 * esize) - UInt(immh:immb);
- boolean round = (op == '1');
- boolean unsigned = (U == '1');

Vector

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 1 1 0 != 0000  immh 1 0 0 1 0 1 Rn Rd

UQSHRN(2) <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
```

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- if immh == '0000' then SFE(asimdimm);
- if immh<3> == '1' then UNDEFINED;
- integer esize = 8 << HighestSetBit(immh);
- integer datasize = 64;
- integer part = UInt(Q);
- integer elements = datasize DIV esize;
- integer shift = (2 * esize) - UInt(immh:immb);
- boolean round = (op == '1');
- boolean unsigned = (U == '1');
**Assembler Symbols**

2. Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>B</td>
</tr>
<tr>
<td>001x</td>
<td>H</td>
</tr>
<tr>
<td>01xx</td>
<td>S</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>H</td>
</tr>
<tr>
<td>001x</td>
<td>S</td>
</tr>
<tr>
<td>01xx</td>
<td>D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(datasize*2) operand = V[n];}
\texttt{bits(datasize) result;}
\texttt{integer round\_const = if round then (1 \ll (shift - 1)) else 0;}
\texttt{integer element;}
\texttt{boolean sat;}
\texttt{for e = 0 to elements-1}
\texttt{element = (Int(Elem(operand, e, 2*esize), unsigned) + round\_const) \gg shift;}
\texttt{(Elem(result, e, esize), sat) = SatQ(element, esize, unsigned);}
\texttt{if sat then FPSR.QC = '1';}
\texttt{Vpart[d, part] = result;}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSUB

Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \( FPSTR.QC \) is set.

Depending on the settings in the \( CPACR\_EL1 \), \( CPTR\_EL2 \), and \( CPTR\_EL3 \) registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

**Scalar**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|.  
| 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | size | 1  | Rd | 0  | 0  | 0  | 0  | 1  | 1  | 1  | Rn | Rd |
| U  |

\[ \text{UQSUB} \langle V \rangle \langle d \rangle, \langle V \rangle \langle n \rangle, \langle V \rangle \langle m \rangle \]

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd) ; \\
\text{integer } n & = \text{UInt}(Rn) ; \\
\text{integer } m & = \text{UInt}(Rm) ; \\
\text{integer } esize & = 8 \ll \text{UInt(size)} ; \\
\text{integer } datasize & = esize ; \\
\text{integer } elements & = 1 ; \\
\text{boolean } unsigned & = (U == '1') ;
\end{align*}
\]

**Vector**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rd | 0  | 0  | 0  | 0  | 1  | 1  | 1  | Rn | Rd |
| U  |

\[ \text{UQSUB} \langle Vd \rangle . \langle T \rangle , \langle Vn \rangle . \langle T \rangle , \langle Vm \rangle . \langle T \rangle \]

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd) ; \\
\text{integer } n & = \text{UInt}(Rn) ; \\
\text{integer } m & = \text{UInt}(Rm) ; \\
\text{if } size:Q == '110' & then \text{UNDEFINED} ; \\
\text{integer } esize & = 8 \ll \text{UInt(size)} ; \\
\text{integer } datasize & = if Q == '1' then 128 else 64 ; \\
\text{integer } elements & = \text{datasize DIV } esize ; \\
\text{boolean } unsigned & = (U == '1') ;
\end{align*}
\]

**Assembler Symbols**

\[ <V> \]

Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\[ <d> \]

Is the number of the SIMD&FP destination register, in the "Rd" field.

\[ <n> \]

Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\[ <m> \]

Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

\[ <Vd> \]

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\[ <T> \]

Is an arrangement specifier, encoded in "size:Q".
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

&lt;Vn&gt;  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

&lt;Vm&gt;  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQXTN, UQXTN2**

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit $FPSR.QC$ is set.

The UQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQXTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the $CPACR_EL1$, $CPTR_EL2$, and $CPTR_EL3$ registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>1 0 1 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

UQXTN $<Vb><d>$, $<Va><n>$

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

### Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>1 0 1 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

UQXTN2 $<Vd>\{2\}<Tb>$, $<Vn>\langle Ta \rangle$

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

### Assembler Symbols

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
</tbody>
</table>

Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

$<Tb>$ Is an arrangement specifier, encoded in “size:Q”: 
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vb> Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Vb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Va&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
boolean sat;
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

 Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
URECPE

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

\[
\begin{array}{ccccccccccccccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 0 & 1 & sz & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & \hline
\end{array}
\]

Rn | Rd

URECPE $<Vd>$.<$T>$, $<Vn>$.<$T>

integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

Assembler Symbols

$<Vd>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<T>$ Is an arrangement specifier, encoded in "sz:Q"

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

$<Vn>$ Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = $V[n]$;
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);
$V[d]$ = result;
URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see UHADD.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 Q 1 0 1 1 0 size 1 Rm 0 0 0 1 0 1 Rn Rd
```

URHADD <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- integer m = UInt(Rm);
- if size == '11' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- boolean unsigned = (U == '1');

Assembler Symbols

- <Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- <T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- <Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- <Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
integer element1;
integer element2;
for e = 0 to elements-1
    element1 = Int(Elem(operand1, e, esize), unsigned);
    element2 = Int(Elem(operand2, e, esize), unsigned);
    Elem[result, e, esize] = (element1+element2+1)<esize:1>;
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
```
URSHL

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>R</td>
<td>S</td>
<td>size</td>
<td>Rd</td>
<td>Rn</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

URSHL `<V><d>`, `<V><n>`, `<V><m>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

**Vector**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>R</td>
<td>S</td>
<td>size</td>
<td>Rd</td>
<td>Rn</td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

URSHL `<Vd>.<T>`, `<Vn>.<T>`, `<Vm>.<T>`

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

**Assembler Symbols**

- **<V>** Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<d>** Is the number of the SIMD&FP destination register, in the "Rd" field.
- **<n>** Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- **<m>** Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
    shift = SInt(Elem(operand2, e, esize)<7:0>);
    if rounding then
        round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        element = (Int(Elem(operand1, e, esize), unsigned) + round_const) << shift;
    if saturating then
        (Elem(result, e, esize), sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = ‘1’;
    else
        Elem(result, e, esize) = element<esize-1:0>;

V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URSHR**

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see **USHR**.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

**Scalar**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0 1 1 1 1 1 1 1 1 0 != 0000</th>
<th>immh</th>
<th>0 0 1 0 0 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

URSHR <V><d>, <V><n>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

**Vector**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0 1 0 1 1 1 1 1 0 != 0000</th>
<th>immh</th>
<th>0 0 1 0 0 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

URSHR <Vd><T>, <Vn><T>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if imm == '0000' then **SEE(asimdimm)**;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << **HighestSetBit**(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

**Assembler Symbols**

```
<V> Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift>

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>shift</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td></td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td></td>
<td>(128- UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>shift</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>(16- UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td></td>
<td>(32- UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td></td>
<td>(64- UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td></td>
<td>(128- UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URSQRT**

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
|   |   |   | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| Q |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if sz == '1' then UNDEFINED;
integer esize = 32;
dataSize = if Q == '1' then 128 else 64;
elements = dataSize DIV esize;

integer esize = 32;
dataSize = if Q == '1' then 128 else 64;
elements = dataSize DIV esize;
```

**CheckFPAdvSIMDEnabled64();**

```
bites(dataSize) operand = V[n];
bites(dataSize) result;
bites(32) element;
for e = 0 to elements-1
  element = Elem[operand, e, 32];
  Elem[result, e, 32] = UnsignedRSqtEstimate(element);
```

V[d] = result;
**URSRA**

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see URSA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: **Scalar** and **Vector**

### Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 1 1 1 1 1 0 ! = 0000 | immh | 0 0 1 1 0 1 | Rn | Rd |
| U | o1 o0 |
```

URSRA `<V>`<`,`<d>`, `<V>`<`,`<n>`, #<shift>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- if immh<3> ! = '1' then UNDEFINED;
- integer esize = 8 << 3;
- integer datasize = esize;
- integer elements = 1;
- integer shift = (esize * 2) - UInt(immh:immb);
- boolean unsigned = (U == '1');
- boolean round = (o1 == '1');
- boolean accumulate = (o0 == '1');

### Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | Q | 1 0 1 1 1 1 0 ! = 0000 | immh | 0 0 1 1 0 1 | Rn | Rd |
| U | o1 o0 |
```

URSRA `<Vd>`<`,`<T>`, `<Vn>`<`,`<T>`, #<shift>

- integer d = UInt(Rd);
- integer n = UInt(Rn);
- if immh == '0000' then SEE(asimdimm);
- if immh<3>:Q == '10' then UNDEFINED;
- integer esize = 8 << HighestSetBit(immh);
- integer datasize = if Q == '1' then 128 else 64;
- integer elements = datasize DIV esize;
- integer shift = (esize * 2) - UInt(immh:immb);
- boolean unsigned = (U == '1');
- boolean round = (o1 == '1');
- boolean accumulate = (o0 == '1');

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>` Is the number of the SIMD&FP destination register, in the "Rd" field.
<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bv(data) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an **OPTIONAL** instruction. *ID_AA64ISAR0_EL1*. I8MM indicates whether this instruction is supported.

### Vector (Armv8.6)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | L  | M  | Rm | 1  | 1  | 1  | 1  | H  | 0  | Rn | Rd |
| US |

USDOT *<Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B*[<index>]*

if ![HaveInt8MatMulExt() then UNDEFINED;](#)

boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

**Assembler Symbols**

*<Vd>*

Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

*<Ta>*

Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

*<Vn>*

Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

*<Tb>*

Is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

*<Vm>*

Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

*<index>*

Is the immediate index of a quadtuplet of four 8-bit elements in the range 0 to 3, encoded in the "H:L" fields.
Operation

CheckFPAdvSIMDEnabled64();
bias(datasize) operand1 = V[n];
bias(128) operand2 = V[m];
bias(datasize) operand3 = V[d];
bias(datasize) result;

for e = 0 to elements-1
    bias(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
        Elem[result, e, 32] = res;
V[d] = result;
**USDOT (vector)**

Dot Product vector form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an **OPTIONAL** instruction. ID_AA64ISAR0_EL1.I8MM indicates whether this instruction is supported.

### Vector (Armv8.6)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
| 0  | Q | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | Rm | 1 | 0 | 0 | 1 | 1 | 1 | Rn | Rd |

**USDOT** <Vd>, <Ta>, <Vn>, <Tb>, <Vm>, <Tb>

if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;

### Assembler Symbols

- **<Vd>** Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
- **<Ta>** Is an arrangement specifier, encoded in "Q":
  - Q | <Ta> |
    - 0 | 25 |
    - 1 | 45 |
- **<Vn>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Tb>** Is an arrangement specifier, encoded in "Q":
  - Q | <Tb> |
    - 0 | 8B |
    - 1 | 16B |
- **<Vm>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

### Operation

```plaintext
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = UInt(Elem[operand1, 4*e+b, 8]);
        integer element2 = SInt(Elem[operand2, 4*e+b, 8]);
        res = res + element1 * element2;
        Elem[result, e, 32] = res;

V[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USHL

Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see USRSHL.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U 0 1 1 1 1 1 0 size 1 Rm 0 1 0 0 0 1 Rn Rd
```

USHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
U 0 1 0 1 1 1 0 size 1 Rm 0 1 0 0 0 1 Rn Rd
```

USHL <Vd>..<T>, <Vn>..<T>, <Vm>..<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

- **<V>**: Is a width specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- **<d>**: Is the number of the SIMD&FP destination register, in the "Rd" field.
- **<n>**: Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
- **<m>**: Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

CheckFPAdvSIMDEnabled64();

```
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer round_const = 0;
integer shift;
integer element;
boolean sat;

for e = 0 to elements-1
  shift = SInt(Elem[operand2, e, esize]<7:0>);
  if rounding then
    round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    element = (Int(Elem[operand1, e, esize], unsigned) + round_const) << shift;
  if saturating then
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';
  else
    Elem[result, e, esize] = element<esize-1:0>;
```

```
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**USHLL, USHLL2**

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The **USHLL** instruction extracts vector elements from the lower half of the source register, while the **USHLL2** instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the **CPACR_EL1**, **CPTR_EL2**, and **CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias **UXTL, UXTL2**.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | !tl  | immh | 1  | 0  | 1  | 0  | 0  | 1  | Rn |    |    |    |    |    |    |    |    |    |    |

**USHLL(2) <Vd>.<Ta>, <Vn>.<Tb>, #<shift>**

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');

**Assembler Symbols**

*2* Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

<table>
<thead>
<tr>
<th>Q</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

*<Vd>* Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

*<Ta>* Is an arrangement specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>01lx</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

*<Vn>* Is the name of the SIMD&FP source register, encoded in the "Rn" field.

*<Tb>* Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01lx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01lx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

*<shift>* Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

USHLL, USHLL2
**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UXTL, UXTL2</td>
<td><code>immb == '000' &amp;&amp; BitCount(immh) == 1</code></td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part];
bits(datasize*2) result;
integer element;
for e = 0 to elements-1
  element = Int(Elem[operand, e, esize], unsigned) << shift;
  Elem[result, e, 2*esize] = element<2*esize-1:0>;
V[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USHR

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see USHR.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 1 1 1 1 1 1 1 0 | != 0000 | immh 0 0 0 0 0 1 | Rd |
|                  |                    |immh                |
| U                |                    |0 1 00              |
```

USHR <V><d>, <V><n>, #<shift>

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 1 0 1 1 1 1 0 | != 0000 | immh 0 0 0 0 0 1 | Rd |
|                  |                    |immh                |
| U                |                    |0 1 00              |
```

USHR <Vd>,<T>, <Vn>,<T>, #<shift>

```python
integer d = UInt(Rd);
integer n = UInt(Rn);
if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

```
<V>   Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<d>   Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in “immh:Q”:

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td>SEE_Advanced_SIMD_modified_immmediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>immb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td></td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td></td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>immb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td>SEE_Advanced_SIMD_modified_immmediate</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td></td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td></td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td></td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USMMLA (vector)

Unsigned and signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an **OPTIONAL** instruction. *ID_AA64ISAR0_EL1*.I8MM indicates whether this instruction is supported.

**Vector (Armv8.6)**

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
| U  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rm | 1  | 0  | 1  | 0  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Rd |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

**Operation**

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];
V[d] = MatMulAdd(addend, operand1, operand2, TRUE, FALSE);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USQADD

Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit FPSR.QC is set.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
|      0 1 1 1 1 1 1 1 0                        | 0 1 0 0 0 0 0 0 0 0 1 1 1 0                  |
|      Rd                                      | Rn                                           |
| U                                            |

USQADD \(<V><d>, <V><n>\)

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');

Vector

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
|      0 0 1 1 1 1 1 1 0                        | 0 1 0 0 0 0 0 0 0 0 1 1 1 0                  |
|      Rd                                      | Rn                                           |
| U                                            |

USQADD \(<Vd>..<T>, <Vn>..<T>\)

integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');

Assembler Symbols

\(<V>\) Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<n>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.

\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in "size:Q":

USQADD
<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(datasize) operand2 = V[d];
integer op1;
integer op2;
boolean sat;
for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see USRRA.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

USRA `<V><d>`, `<V><n>`, `#<shift>`

integer d = UInt(Rd);
integer n = UInt(Rn);
if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 1 0 1 1 1 1 0</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

USRA `<Vd>,<T>, <Vn>..<T>, #<shift>`

integer d = UInt(Rd);
integer n = UInt(Rn);
if imm == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

`<V>` Is a width specifier, encoded in “immh”:

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

`<d>` Is the number of the SIMD&FP destination register, in the “Rd” field.
Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0 4H</td>
</tr>
<tr>
<td>001x</td>
<td>1 8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0 25</td>
</tr>
<tr>
<td>01xx</td>
<td>1 4S</td>
</tr>
<tr>
<td>1xx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xx</td>
<td>1 20</td>
</tr>
</tbody>
</table>

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in "immh:immb":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
</tr>
<tr>
<td>0001</td>
<td>(16-UInt(immh:immb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
operand2 = if accumulate then V[d] else Zeros();
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register, while the USUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

USUBL2 {2} <Vd>, <Ta>, <Vn>, <Vm>, <Tb>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

| 0 | 2 |
|----|
| 0 | [absent] |
| 1 | [present] |

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USUBW, USUBW2

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source register, while the USUBW2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 | Q | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 1 | 1 | 0 | 0 | Rn | Rd |
   U  

USUBW2 <Vd>..<Ta>, <Vn>..<Vn>, <Vm>..<Tb>
```

Integer d = UInt(Rd);
Integer n = UInt(Rn);
Integer m = UInt(Rm);

If size == '11' then UNDEFINED;
Integer esize = 8 << UInt(size);
Integer datasize = 64;
Integer part = UInt(Q);
Integer elements = datasize DIV esize;

Boolean sub_op = (o1 == '1');
Boolean unsigned = (U == '1');

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":

```
Q 2
 0 [absent]
 1 [present]
```

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

```
size <Ta>
 00 8H
 01 4S
 10 2D
 11 RESERVED
```

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

```
size Q <Tb>
 00 0 8B
 00 1 16B
 01 0 4H
 01 1 8H
 10 0 2S
 10 1 4S
 11 x RESERVED
```
Operation

CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
  element1 = Int(Elem[operand1, e, 2*esize], unsigned);
  element2 = Int(Elem[operand2, e, esize], unsigned);
  if sub_op then
    sum = element1 - element2;
  else
    sum = element1 + element2;
  Elem[result, e, 2*esize] = sum<2*esize-1:0>;
V[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
**UXTL, UXTL2**

Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. The UXTL instruction extracts vector elements from the lower half of the source register, while the UXTL2 instruction extracts vector elements from the upper half of the source register. Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of **USHLL, USHLL2**. This means:

- The encodings in this description are named to match the encodings of **USHLL, USHLL2**.
- The description of **USHLL, USHLL2** gives the operational pseudocode for this instruction.

![Register Encoding](image)

UXTL{2} <Vd>.<Ta>, <Vn>.<Tb>

is equivalent to

USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when BitCount(immh) == 1.

**Assembler Symbols**

- **Q**
  - 0 [absent]
  - 1 [present]

- **<Vd>**
  - Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

- **<Ta>**
  - Is an arrangement specifier, encoded in "immh":

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- **<Vn>**
  - Is the name of the SIMD&FP source register, encoded in the "Rn" field.

- **<Tb>**
  - Is an arrangement specifier, encoded in "immh:Q":

<table>
<thead>
<tr>
<th>immh</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

The description of **USHLL, USHLL2** gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DI T is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UZP1**

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can be used with UZP2 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.

![Example of UZP1 and UZP2 operation]

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UZP2**

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination SIMD&FP register.

This instruction can be used with UZP1 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.

![Example of UZP1 and UZP2 operation](image)

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];
V[d] = result;

Operational information

If PSTATE.DIT is 1:
    • The execution time of this instruction is independent of:
        ◦ The values of the data supplied in any of its registers.
        ◦ The values of the NZCV flags.
    • The response of this instruction to asynchronous exceptions does not vary based on:
        ◦ The values of the data supplied in any of its registers.
        ◦ The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Exclusive OR and Rotate performs a bitwise exclusive OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when \textit{FEAT\_SHA3} is implemented.

**Advanced SIMD**

(Armv8.2)

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| Rd |     |     | Rm |     | imm6 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>

if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

**Assembler Symbols**

- \texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- \texttt{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- \texttt{<Vm>} Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
- \texttt{<imm6>} Is a rotation right, encoded in "imm6".

**Operation**

\texttt{AArch64.CheckFPAdvSIMDEnabled();}

\texttt{bits(128) Vm = V[m];}
\texttt{bits(128) Vn = V[n];}
\texttt{bits(128) tmp;}
\texttt{tmp = Vn EOR Vm;}
\texttt{V[d] = ROR(tmp<127:64>, UInt(imm6)):ROR(tmp<63:0>, UInt(imm6));}

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**XTN, XTN2**

Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The XTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the XTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1, CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 0  | size | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | Rn | | | | | | | | | | | | | | Rd |

**XTN{2} <Vd>,<Tb>, <Vn>,<Ta>**

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>absent</td>
<td>present</td>
</tr>
</tbody>
</table>

**<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

**<Tb>** Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**<Vn>** Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**<Ta>** Is an arrangement specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8H</td>
</tr>
<tr>
<td>01</td>
<td>4S</td>
</tr>
<tr>
<td>10</td>
<td>2D</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
```

```c
bits(2*datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) element;
```

```c
for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
    Vpart[d, part] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ZIP1

Zip vectors (primary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with ZIP2 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

```
< Vd > < T >, < Vm > < T >, < Vm > < T >
```

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);
integer \( m = \text{UInt}(Rm) \);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = \text{UInt}(op);
integer pairs = elements DIV 2;

**Assembler Symbols**

- **<Vd>** Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- **<T>** Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- **<Vm>** Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- **<Vn>** Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];
V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Zip vectors (secondary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

This instruction can be used with ZIP1 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

```
Vn | A7 A6 A5 A4 A3 A2 A1 A0 |
Vm | B7 B6 B5 B4 B3 B2 B1 B0 |
```

Depending on the settings in the **CPACR_EL1, CPTR_EL2, and CPTR_EL3** registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

```
0 0 0 0 1 1 1 0 | size:Q | 0 1 1 1 1 0 | Rm | Rd
```

**Assembler Symbols**

- `<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>` Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>

- `<Vn>` Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
- `<Vm>` Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
  Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
  Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
A64 -- SVE Instructions (alphabetic order)

**ABS**: Absolute value (predicated).

**ADCLB**: Add with carry long (bottom).

**ADCLT**: Add with carry long (top).

**ADD (immediate)**: Add immediate (unpredicated).

**ADD (vectors, predicated)**: Add vectors (predicated).

**ADD (vectors, unpredicated)**: Add vectors (unpredicated).

**ADDHNB**: Add narrow high part (bottom).

**ADDHNT**: Add narrow high part (top).

**ADDP**: Add pairwise.

**ADDPL**: Add multiple of predicate register size to scalar register.

**ADDVL**: Add multiple of vector register size to scalar register.

**ADR**: Compute vector address.

**AESD**: AES single round decryption.

**AESE**: AES single round encryption.

**AESIMC**: AES inverse mix columns.

**AESMC**: AES mix columns.

**AND (immediate)**: Bitwise AND with immediate (unpredicated).

**AND (vectors, predicated)**: Bitwise AND vectors (predicated).

**AND (vectors, unpredicated)**: Bitwise AND vectors (unpredicated).

**AND, ANDS (predicates)**: Bitwise AND predicates.

**ANDV**: Bitwise AND reduction to scalar.

**ASR (immediate, predicated)**: Arithmetic shift right by immediate (predicated).

**ASR (immediate, unpredicated)**: Arithmetic shift right by immediate (unpredicated).

**ASR (vectors)**: Arithmetic shift right by vector (predicated).

**ASR (wide elements, predicated)**: Arithmetic shift right by 64-bit wide elements (predicated).

**ASR (wide elements, unpredicated)**: Arithmetic shift right by 64-bit wide elements (unpredicated).

**ASRD**: Arithmetic shift right for divide by immediate (predicated).

**ASRR**: Reversed arithmetic shift right by vector (predicated).

**BCAX**: Bitwise clear and exclusive OR.

**BDEP**: Scatter lower bits into positions selected by bitmask.

**BEXT**: Gather lower bits from positions selected by bitmask.

**BFCVT**: Floating-point down convert to BFloat16 format (predicated).

**BFCVNT**: Floating-point down convert and narrow to BFloat16 (top, predicated).

**BFDOT (indexed)**: BFloat16 floating-point indexed dot product.
BFDOT (vectors): BFloat16 floating-point dot product.

BFMLALB (indexed): BFloat16 floating-point multiply-add long to single-precision (bottom, indexed).

BFMLALB (vectors): BFloat16 floating-point multiply-add long to single-precision (bottom).

BFMLALT (indexed): BFloat16 floating-point multiply-add long to single-precision (top, indexed).

BFMLALT (vectors): BFloat16 floating-point multiply-add long to single-precision (top).

BFMMLA: BFloat16 floating-point matrix multiply-accumulate.

BGRP: Group bits to right or left as selected by bitmask.

BIC (immediate): Bitwise clear bits using immediate (unpredicated): an alias of AND (immediate).

BIC (vectors, predicated): Bitwise clear vectors (predicated).

BIC (vectors, unpredicated): Bitwise clear vectors (unpredicated).

BIC, BICS (predicates): Bitwise clear predicates.

BRKA, BRKAS: Break after first true condition.

BRKB, BRKBS: Break before first true condition.

BRKN, BRKNS: Propagate break to next partition.

BRKPA, BRKPAS: Break after first true condition, propagating from previous partition.

BRKPB, BRKPBS: Break before first true condition, propagating from previous partition.

BSL: Bitwise select.

BSL1N: Bitwise select with first input inverted.

BSL2N: Bitwise select with second input inverted.

CADD: Complex integer add with rotate.

CDOT (indexed): Complex integer dot product (indexed).

CDOT (vectors): Complex integer dot product.

CLASTA (scalar): Conditionally extract element after last to general-purpose register.

CLASTA (SIMD&FP scalar): Conditionally extract element after last to SIMD&FP scalar register.

CLASTA (vectors): Conditionally extract element after last to vector register.

CLASTB (scalar): Conditionally extract last element to general-purpose register.

CLASTB (SIMD&FP scalar): Conditionally extract last element to SIMD&FP scalar register.

CLASTB (vectors): Conditionally extract last element to vector register.

CLS: Count leading sign bits (predicated).

CLZ: Count leading zero bits (predicated).

CMLA (indexed): Complex integer multiply-add with rotate (indexed).

CMLA (vectors): Complex integer multiply-add with rotate.

CMP<cc> (immediate): Compare vector to immediate.

CMP<cc> (vectors): Compare vectors.

CMP<cc> (wide elements): Compare vector to 64-bit wide elements.

CMPEQ (vectors): Compare signed less than or equal to vector, setting the condition flags: an alias of CMP<cc> (vectors).
CMPLO (vectors): Compare unsigned lower than vector, setting the condition flags: an alias of CMP<cc> (vectors).

CMPLS (vectors): Compare unsigned lower or same as vector, setting the condition flags: an alias of CMP<cc> (vectors).

CMPLT (vectors): Compare signed less than vector, setting the condition flags: an alias of CMP<cc> (vectors).

CNOT: Logically invert boolean condition in vector (predicated).

CNT: Count non-zero bits (predicated).

CNTB, CNTD, CNTH, CNTW: Set scalar to multiple of predicate constraint element count.

CNTP: Set scalar to count of true predicate elements.

COMPACT: Shuffle active elements of vector to the right and fill with zero.

CPY (immediate, merging): Copy signed integer immediate to vector elements (merging).

CPY (immediate, zeroing): Copy signed integer immediate to vector elements (zeroing).

CPY (scalar): Copy general-purpose register to vector elements (predicated).

CPY (SIMD&FP scalar): Copy SIMD&FP scalar register to vector elements (predicated).

CTERMEQ, CTERMNE: Compare and terminate loop.

DECB, DECD, DECH, DECW (scalar): Decrement scalar by multiple of predicate constraint element count.

DECD, DECH, DECW (vector): Decrement vector by multiple of predicate constraint element count.

DECP (scalar): Decrement scalar by count of true predicate elements.

DECP (vector): Decrement vector by count of true predicate elements.

DUP (immediate): Broadcast signed immediate to vector elements (unpredicated).

DUP (indexed): Broadcast indexed element to vector (unpredicated).

DUP (scalar): Broadcast general-purpose register to vector elements (unpredicated).

DUPM: Broadcast logical bitmask immediate to vector (unpredicated).

EON: Bitwise exclusive OR with inverted immediate (unpredicated): an alias of EOR (immediate).

EOR (immediate): Bitwise exclusive OR with immediate (unpredicated).

EOR (vectors, predicated): Bitwise exclusive OR vectors (predicated).

EOR (vectors, unpredicated): Bitwise exclusive OR vectors (unpredicated).

EOR, EORS (predicates): Bitwise exclusive OR predicates.

EOR3: Bitwise exclusive OR of three vectors.

EORBT: Interleaving exclusive OR (bottom, top).

EORTB: Interleaving exclusive OR (top, bottom).

EORV: Bitwise exclusive OR reduction to scalar.

EXT: Extract vector from pair of vectors.

FABD: Floating-point absolute difference (predicated).

FABS: Floating-point absolute value (predicated).

FAC<cc>: Floating-point absolute compare vectors.

FACLE: Floating-point absolute compare less than or equal: an alias of FAC<cc>.

FACLT: Floating-point absolute compare less than: an alias of FAC<cc>.
**FADD (immediate)**: Floating-point add immediate (predicated).

**FADD (vectors, predicated)**: Floating-point add vector (predicated).

**FADD (vectors, unpredicated)**: Floating-point add vector (unpredicated).

**FADDA**: Floating-point add strictly-ordered reduction, accumulating in scalar.

**FADDP**: Floating-point add pairwise.

**FADDV**: Floating-point add recursive reduction to scalar.

**FCADD**: Floating-point complex add with rotate (predicated).

**FCM<cc> (vectors)**: Floating-point compare vectors.

**FCM<cc> (zero)**: Floating-point compare vector with zero.

**FCMLA (indexed)**: Floating-point complex multiply-add by indexed values with rotate.

**FCMLA (vectors)**: Floating-point complex multiply-add with rotate (predicated).

**FCMLE (vectors)**: Floating-point compare less than or equal to vector: an alias of FCM<cc> (vectors).

**FCMLT (vectors)**: Floating-point compare less than vector: an alias of FCM<cc> (vectors).

**FCPY**: Copy 8-bit floating-point immediate to vector elements (predicated).

**FCVT**: Floating-point convert precision (predicated).

**FCVTLT**: Floating-point up convert long (top, predicated).

**FCVTNT**: Floating-point down convert and narrow (top, predicated).

**FCVTX**: Floating-point down convert, rounding to odd (predicated).

**FCVTXNT**: Floating-point down convert, rounding to odd (top, predicated).

**FCVTZS**: Floating-point convert to signed integer, rounding toward zero (predicated).

**FCVTZU**: Floating-point convert to unsigned integer, rounding toward zero (predicated).

**FDIV**: Floating-point divide by vector (predicated).

**FDIVR**: Floating-point reversed divide by vector (predicated).

**FDUP**: Broadcast 8-bit floating-point immediate to vector elements (unpredicated).

**FEXPA**: Floating-point exponential accelerator.

**FLOGR**: Floating-point base 2 logarithm as integer.

**FMAD**: Floating-point fused multiply-add vectors (predicated), writing multiplicand \[Zdn = Za + Zdn * Zm\].

**FMAX (immediate)**: Floating-point maximum with immediate (predicated).

**FMAX (vectors)**: Floating-point maximum with immediate (predicated).

**FMAXNM (immediate)**: Floating-point maximum number with immediate (predicated).

**FMAXNM (vectors)**: Floating-point maximum number (predicated).

**FMAXNMP**: Floating-point maximum number pairwise.

**FMAXNMV**: Floating-point maximum number recursive reduction to scalar.

**FMAXP**: Floating-point maximum pairwise.

**FMAXV**: Floating-point maximum recursive reduction to scalar.

**FMIN (immediate)**: Floating-point minimum with immediate (predicated).
FMIN (vectors): Floating-point minimum (predicated).
FMINNM (immediate): Floating-point minimum number with immediate (predicated).
FMINNM (vectors): Floating-point minimum number (predicated).
FMINNMP: Floating-point minimum number pairwise.
FMINNNMV: Floating-point minimum number recursive reduction to scalar.
FMINP: Floating-point minimum pairwise.
FMINV: Floating-point minimum recursive reduction to scalar.
FMLA (indexed): Floating-point fused multiply-add by indexed elements (Zda = Zda + Zn * Zm[indexed]).
FMLA (vectors): Floating-point fused multiply-add vectors (predicated), writing addend [Zda = Zda + Zn * Zm].
FMLALB (indexed): Half-precision floating-point multiply-add long to single-precision (bottom, indexed).
FMLALB (vectors): Half-precision floating-point multiply-add long to single-precision (bottom).
FMLALT (indexed): Half-precision floating-point multiply-add long to single-precision (top, indexed).
FMLALT (vectors): Half-precision floating-point multiply-add long to single-precision (top).
FMLSL (indexed): Floating-point fused multiply-subtract by indexed elements (Zda = Zda + -Zn * Zm[indexed]).
FMLSL (vectors): Floating-point fused multiply-subtract vectors (predicated), writing addend [Zda = Zda + -Zn * Zm].
FMLSLB (indexed): Half-precision floating-point multiply-subtract long from single-precision (bottom, indexed).
FMLSLB (vectors): Half-precision floating-point multiply-subtract long from single-precision (bottom).
FMLSLT (indexed): Half-precision floating-point multiply-subtract long from single-precision (top, indexed).
FMLSLT (vectors): Half-precision floating-point multiply-subtract long from single-precision (top).
FMMLA: Floating-point matrix multiply-accumulate.
FMOV (immediate, predicated): Move 8-bit floating-point immediate to vector elements (predicated): an alias of FCPY.
FMOV (immediate, unpredicated): Move 8-bit floating-point immediate to vector elements (unpredicated): an alias of FDUP.
FMOV (zero, predicated): Move floating-point +0.0 to vector elements (predicated): an alias of CPY (immediate, merging).
FMOV (zero, unpredicated): Move floating-point +0.0 to vector elements (unpredicated): an alias of DUP (immediate).
FMSB: Floating-point fused multiply-subtract vectors (predicated), writing multiplicand [Zdn = Za + -Zdn * Zm].
FMUL (immediate): Floating-point multiply by immediate (predicated).
FMUL (indexed): Floating-point multiply by indexed elements.
FMUL (vectors, predicated): Floating-point multiply vectors (predicated).
FMUL (vectors, unpredicated): Floating-point multiply vectors (unpredicated).
FMULX: Floating-point multiply-extended vectors (predicated).
FNEG: Floating-point negate (predicated).
FNMA: Floating-point negated fused multiply-add vectors (predicated), writing multiplicand [Zdn = -Za + -Zdn * Zm].
FNMLA: Floating-point negated fused multiply-add vectors (predicated), writing addend [Zda = -Zda + -Zn * Zm].
FNMLS: Floating-point negated fused multiply-subtract vectors (predicated), writing addend [Zda = -Zda + Zn * Zm].
**FNMSB**: Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand \( Z_{dn} = -Z_a + Z_{dn} * Z_m \).

**FRECPE**: Floating-point reciprocal estimate (unpredicated).

**FRECPS**: Floating-point reciprocal step (unpredicated).

**FRECPX**: Floating-point reciprocal exponent (predicated).

**PRINT<\text{r}>**: Floating-point round to integral value (predicated).

**FRSORTE**: Floating-point reciprocal square root estimate (unpredicated).

**FRSORTS**: Floating-point reciprocal square root step (unpredicated).

**FSCALE**: Floating-point adjust exponent by vector (predicated).

**FSORT**: Floating-point square root (predicated).

**FSUB (immediate)**: Floating-point subtract immediate (predicated).

**FSUB (vectors, predicated)**: Floating-point subtract vectors (predicated).

**FSUB (vectors, unpredicated)**: Floating-point subtract vectors (unpredicated).

**FSUBR (immediate)**: Floating-point reversed subtract from immediate (predicated).

**FSUBR (vectors)**: Floating-point reversed subtract vectors (predicated).

**FTMAD**: Floating-point trigonometric multiply-add coefficient.

**FTSMUL**: Floating-point trigonometric starting value.

**FTSSEL**: Floating-point trigonometric select coefficient.

**HISTCNT**: Count matching elements in vector.

**HISTSEG**: Count matching elements in vector segments.

**INCB, INCD, INCH, INCW (scalar)**: Increment scalar by multiple of predicate constraint element count.

**INCD, INCH, INCW (vector)**: Increment vector by multiple of predicate constraint element count.

**INCP (scalar)**: Increment scalar by count of true predicate elements.

**INCP (vector)**: Increment vector by count of true predicate elements.

**INDEX (immediate, scalar)**: Create index starting from immediate and incremented by general-purpose register.

**INDEX (immediates)**: Create index starting from and incremented by immediate.

**INDEX (scalar, immediate)**: Create index starting from general-purpose register and incremented by immediate.

**INDEX (scalars)**: Create index starting from and incremented by general-purpose register.

**INSR (scalar)**: Insert general-purpose register in shifted vector.

**INSR (SIMD&FP scalar)**: Insert SIMD&FP scalar register in shifted vector.

**LASTA (scalar)**: Extract element after last to general-purpose register.

**LASTA (SIMD&FP scalar)**: Extract element after last to SIMD&FP scalar register.

**LASTB (scalar)**: Extract last element to general-purpose register.

**LASTB (SIMD&FP scalar)**: Extract last element to SIMD&FP scalar register.

**LD1B (scalar plus immediate)**: Contiguous load unsigned bytes to vector (immediate index).

**LD1B (scalar plus scalar)**: Contiguous load unsigned bytes to vector (scalar index).

**LD1B (scalar plus vector)**: Gather load unsigned bytes to vector (vector index).
LD1B (vector plus immediate): Gather load unsigned bytes to vector (immediate index).
LD1D (scalar plus immediate): Contiguous load doublewords to vector (immediate index).
LD1D (scalar plus scalar): Contiguous load doublewords to vector (scalar index).
LD1D (vector plus vector): Gather load doublewords to vector (vector index).
LD1D (vector plus immediate): Gather load doublewords to vector (immediate index).
LD1H (scalar plus immediate): Contiguous load unsigned halfwords to vector (immediate index).
LD1H (scalar plus scalar): Contiguous load unsigned halfwords to vector (scalar index).
LD1H (scalar plus vector): Gather load unsigned halfwords to vector (vector index).
LD1H (vector plus immediate): Gather load unsigned halfwords to vector (immediate index).
LD1RB: Load and broadcast unsigned byte to vector.
LD1RD: Load and broadcast doubleword to vector.
LD1RH: Load and broadcast unsigned halfword to vector.
LD1ROB (scalar plus immediate): Contiguous load and replicate thirty-two bytes (immediate index).
LD1ROB (scalar plus scalar): Contiguous load and replicate thirty-two bytes (scalar index).
LD1ROD (scalar plus immediate): Contiguous load and replicate four doublewords (immediate index).
LD1ROD (scalar plus scalar): Contiguous load and replicate four doublewords (scalar index).
LD1ROH (scalar plus immediate): Contiguous load and replicate sixteen halfwords (immediate index).
LD1ROH (scalar plus scalar): Contiguous load and replicate sixteen halfwords (scalar index).
LD1ROW (scalar plus immediate): Contiguous load and replicate eight words (immediate index).
LD1ROW (scalar plus scalar): Contiguous load and replicate eight words (scalar index).
LD1ROB (scalar plus immediate): Contiguous load and replicate sixteen bytes (immediate index).
LD1ROB (scalar plus scalar): Contiguous load and replicate sixteen bytes (scalar index).
LD1RD (scalar plus immediate): Contiguous load and replicate two doublewords (immediate index).
LD1RD (scalar plus scalar): Contiguous load and replicate two doublewords (scalar index).
LD1ROH (scalar plus immediate): Contiguous load and replicate eight halfwords (immediate index).
LD1ROH (scalar plus scalar): Contiguous load and replicate eight halfwords (scalar index).
LD1ROW (scalar plus immediate): Contiguous load and replicate four words (immediate index).
LD1ROW (scalar plus scalar): Contiguous load and replicate four words (scalar index).
LD1RSB: Load and broadcast signed byte to vector.
LD1RSH: Load and broadcast signed halfword to vector.
LD1RSW: Load and broadcast signed word to vector.
LD1RW: Load and broadcast unsigned word to vector.
LD1SB (scalar plus immediate): Contiguous load signed bytes to vector (immediate index).
LD1SB (scalar plus scalar): Contiguous load signed bytes to vector (scalar index).
LD1SB (scalar plus vector): Gather load signed bytes to vector (vector index).
LD1SB (vector plus immediate): Gather load signed bytes to vector (immediate index).
LD1SH (scalar plus immediate): Contiguous load signed halfwords to vector (immediate index).
LD1SH (scalar plus scalar): Contiguous load signed halfwords to vector (scalar index).
LD1SH (scalar plus vector): Gather load signed halfwords to vector (vector index).
LD1SH (vector plus immediate): Gather load signed halfwords to vector (immediate index).
LD1SW (scalar plus immediate): Contiguous load signed words to vector (immediate index).
LD1SW (scalar plus scalar): Contiguous load signed words to vector (scalar index).
LD1SW (scalar plus vector): Gather load signed words to vector (vector index).
LD1SW (vector plus immediate): Gather load signed words to vector (immediate index).
LD1W (scalar plus immediate): Contiguous load unsigned words to vector (immediate index).
LD1W (scalar plus scalar): Contiguous load unsigned words to vector (scalar index).
LD1W (vector plus immediate): Gather load unsigned words to vector (immediate index).
LD1W (vector plus scalar): Gather load unsigned words to vector (vector index).
LD2B (scalar plus immediate): Contiguous load two-byte structures to two vectors (immediate index).
LD2B (scalar plus scalar): Contiguous load two-byte structures to two vectors (scalar index).
LD2D (scalar plus immediate): Contiguous load two-doubleword structures to two vectors (immediate index).
LD2D (scalar plus scalar): Contiguous load two-doubleword structures to two vectors (scalar index).
LD2H (scalar plus immediate): Contiguous load two-halfword structures to two vectors (immediate index).
LD2H (scalar plus scalar): Contiguous load two-halfword structures to two vectors (scalar index).
LD2W (scalar plus immediate): Contiguous load two-word structures to two vectors (immediate index).
LD2W (scalar plus scalar): Contiguous load two-word structures to two vectors (scalar index).
LD3B (scalar plus immediate): Contiguous load three-byte structures to three vectors (immediate index).
LD3B (scalar plus scalar): Contiguous load three-byte structures to three vectors (scalar index).
LD3D (scalar plus immediate): Contiguous load three-doubleword structures to three vectors (immediate index).
LD3D (scalar plus scalar): Contiguous load three-doubleword structures to three vectors (scalar index).
LD3H (scalar plus immediate): Contiguous load three-halfword structures to three vectors (immediate index).
LD3H (scalar plus scalar): Contiguous load three-halfword structures to three vectors (scalar index).
LD3W (scalar plus immediate): Contiguous load three-word structures to three vectors (immediate index).
LD3W (scalar plus scalar): Contiguous load three-word structures to three vectors (scalar index).
LD4B (scalar plus immediate): Contiguous load four-byte structures to four vectors (immediate index).
LD4B (scalar plus scalar): Contiguous load four-byte structures to four vectors (scalar index).
LD4D (scalar plus immediate): Contiguous load four-doubleword structures to four vectors (immediate index).
LD4D (scalar plus scalar): Contiguous load four-doubleword structures to four vectors (scalar index).
LD4H (scalar plus immediate): Contiguous load four-halfword structures to four vectors (immediate index).
LD4H (scalar plus scalar): Contiguous load four-halfword structures to four vectors (scalar index).
LD4W (scalar plus immediate): Contiguous load four-word structures to four vectors (immediate index).
LD4W (scalar plus scalar): Contiguous load four-word structures to four vectors (scalar index).
LDFF1B (scalar plus scalar): Contiguous load first-fault unsigned bytes to vector (scalar index).
LDFF1B (scalar plus vector): Gather load first-fault unsigned bytes to vector (vector index).
LDFF1B (vector plus immediate): Gather load first-fault unsigned bytes to vector (immediate index).
LDFF1D (scalar plus scalar): Contiguous load first-fault doublewords to vector (scalar index).
LDFF1D (scalar plus vector): Gather load first-fault doublewords to vector (vector index).
LDFF1D (vector plus immediate): Gather load first-fault doublewords to vector (immediate index).
LDFF1H (scalar plus scalar): Contiguous load first-fault unsigned halfwords to vector (scalar index).
LDFF1H (scalar plus vector): Gather load first-fault unsigned halfwords to vector (vector index).
LDFF1H (vector plus immediate): Gather load first-fault unsigned halfwords to vector (immediate index).
LDFF1SB (scalar plus scalar): Contiguous load first-fault signed bytes to vector (scalar index).
LDFF1SB (scalar plus vector): Gather load first-fault signed bytes to vector (vector index).
LDFF1SB (vector plus immediate): Gather load first-fault signed bytes to vector (immediate index).
LDFF1SH (scalar plus scalar): Contiguous load first-fault signed halfwords to vector (scalar index).
LDFF1SH (scalar plus vector): Gather load first-fault signed halfwords to vector (vector index).
LDFF1SH (vector plus immediate): Gather load first-fault signed halfwords to vector (immediate index).
LDFF1SW (scalar plus scalar): Contiguous load first-fault signed words to vector (scalar index).
LDFF1SW (scalar plus vector): Gather load first-fault signed words to vector (vector index).
LDFF1SW (vector plus immediate): Gather load first-fault signed words to vector (immediate index).
LDFF1W (scalar plus scalar): Contiguous load first-fault unsigned words to vector (scalar index).
LDFF1W (scalar plus vector): Gather load first-fault unsigned words to vector (vector index).
LDFF1W (vector plus immediate): Gather load first-fault unsigned words to vector (immediate index).
LDNF1B: Contiguous load non-fault unsigned bytes to vector (immediate index).
LDNF1D: Contiguous load non-fault doublewords to vector (immediate index).
LDNF1H: Contiguous load non-fault unsigned halfwords to vector (immediate index).
LDNF1SB: Contiguous load non-fault signed bytes to vector (immediate index).
LDNF1SH: Contiguous load non-fault signed halfwords to vector (immediate index).
LDNF1SW: Contiguous load non-fault signed words to vector (immediate index).
LDNF1W: Contiguous load non-fault unsigned words to vector (immediate index).
LDNT1B (scalar plus immediate): Contiguous load non-temporal bytes to vector (immediate index).
LDNT1B (scalar plus scalar): Contiguous load non-temporal bytes to vector (scalar index).
LDNT1B (vector plus scalar): Gather load non-temporal unsigned bytes.
LDNT1D (scalar plus immediate): Contiguous load non-temporal doublewords to vector (immediate index).
LDNT1D (scalar plus scalar): Contiguous load non-temporal doublewords to vector (scalar index).
LDNT1D (vector plus scalar): Gather load non-temporal unsigned doublewords.
LDNT1H (scalar plus immediate): Contiguous load non-temporal halfwords to vector (immediate index).
LDNT1H (scalar plus scalar): Contiguous load non-temporal halfwords to vector (scalar index).
LDNT1H (vector plus scalar): Gather load non-temporal unsigned halfwords.
LDNT1SB: Gather load non-temporal signed bytes.
LDNT1SH: Gather load non-temporal signed halfwords.
LDNT1SW: Gather load non-temporal signed words.
LDNT1W (scalar plus immediate): Contiguous load non-temporal words to vector (immediate index).
LDNT1W (scalar plus scalar): Contiguous load non-temporal words to vector (scalar index).
LDNT1W (vector plus scalar): Gather load non-temporal unsigned words.
LDR (predicate): Load predicate register.
LDR (vector): Load vector register.
LSL (immediate, predicated): Logical shift left by immediate (predicated).
LSL (immediate, unpredicated): Logical shift left by immediate (unpredicated).
LSL (vectors): Logical shift left by vector (predicated).
LSL (wide elements, predicated): Logical shift left by 64-bit wide elements (predicated).
LSL (wide elements, unpredicated): Logical shift left by 64-bit wide elements (unpredicated).
LSLR: Reversed logical shift left by vector (predicated).
LSR (immediate, predicated): Logical shift right by immediate (predicated).
LSR (immediate, unpredicated): Logical shift right by immediate (unpredicated).
LSR (vectors): Logical shift right by vector (predicated).
LSR (wide elements, predicated): Logical shift right by 64-bit wide elements (predicated).
LSR (wide elements, unpredicated): Logical shift right by 64-bit wide elements (unpredicated).
LSRR: Reversed logical shift right by vector (predicated).
MAD: Multiply-add vectors (predicated), writing multiplicand \([Zdn = Za + Zdn \times Zm]\).
MATCH: Detect any matching elements, setting the condition flags.
MLA (indexed): Multiply-add to accumulator (indexed).
MLA (vectors): Multiply-add vectors (predicated), writing addend \([Zda = Zda + Zn \times Zm]\).
MLS (indexed): Multiply-subtract from accumulator (indexed).
MLS (vectors): Multiply-subtract vectors (predicated), writing addend \([Zda = Zda - Zn \times Zm]\).
MOV (bitmask immediate): Move logical bitmask immediate to vector (unpredicated): an alias of DUPM.
MOV (immediate, predicated, merging): Move signed integer immediate to vector elements (merging): an alias of CPY (immediate, merging).
MOV (immediate, predicated, zeroing): Move signed integer immediate to vector elements (zeroing): an alias of CPY (immediate, zeroing).
MOV (immediate, unpredicated): Move signed immediate to vector elements (unpredicated): an alias of DUP (immediate).
MOV (scalar, predicated): Move general-purpose register to vector elements (predicated): an alias of CPY (scalar).
**MOV (scalar, unpredicated)**: Move general-purpose register to vector elements (unpredicated): an alias of DUP (scalar).

**MOV (SIMD&FP scalar, predicated)**: Move SIMD&FP scalar register to vector elements (predicated): an alias of CPY (SIMD&FP scalar).

**MOV (SIMD&FP scalar, unpredicated)**: Move indexed element or SIMD&FP scalar to vector (unpredicated): an alias of DUP (indexed).

**MOV (vector, predicated)**: Move vector elements (predicated): an alias of SEL (vectors).

**MOV (vector, unpredicated)**: Move vector register (unpredicated): an alias of ORR (vectors, unpredicated).

**MOVPREFIX (predicated)**: Move prefix (predicated).

**MOVPREFIX (unpredicated)**: Move prefix (unpredicated).

**MOVS (predicated)**: Move predicates (zeroing), setting the condition flags: an alias of AND, ANDS (predicates).

**MOVS (unpredicated)**: Move predicate (unpredicated), setting the condition flags: an alias of ORR, ORRS (predicates).

**MSB**: Multiply-subtract vectors (predicated), writing multiplicand \( Z_{dn} = Z_a - Z_{dn} \times Z_m \).

**MUL (immediate)**: Multiply by immediate (unpredicated).

**MUL (indexed)**: Multiply (indexed).

**MUL (vectors, predicated)**: Multiply vectors (predicated).

**MUL (vectors, unpredicated)**: Multiply vectors (unpredicated).

**NAND, NANDS**: Bitwise NAND predicates.

**NBSL**: Bitwise inverted select.

**NEG**: Negate (predicated).

**NMATCH**: Detect no matching elements, setting the condition flags.

**NOR, NOR**: Bitwise NOR predicates.

**NOT (predicate)**: Bitwise invert predicate: an alias of EOR, EORS (predicates).

**NOT (vector)**: Bitwise invert vector (predicated).

**NOTS**: Bitwise invert predicate, setting the condition flags: an alias of EOR, EORS (predicates).

**ORN (immediate)**: Bitwise inclusive OR with inverted immediate (unpredicated): an alias of ORR (immediate).

**ORN, ORNS (predicates)**: Bitwise inclusive OR inverted predicate.

**ORR (immediate)**: Bitwise inclusive OR with immediate (unpredicated).

**ORR (vectors, predicated)**: Bitwise inclusive OR vectors (predicated).

**ORR (vectors, unpredicated)**: Bitwise inclusive OR vectors (unpredicated).

**ORR, ORRS (predicates)**: Bitwise inclusive OR predicate.

**ORV**: Bitwise inclusive OR reduction to scalar.

**PFALSE**: Set all predicate elements to false.

**PFIRST**: Set the first active predicate element to true.

**PMUL**: Polynomial multiply vectors (unpredicated).

**PMULLB**: Polynomial multiply long (bottom).

**PMULLT**: Polynomial multiply long (top).

**PNEXT**: Find next active predicate.
PRFB (scalar plus immediate): Contiguous prefetch bytes (immediate index).
PRFB (scalar plus scalar): Contiguous prefetch bytes (scalar index).
PRFB (scalar plus vector): Gather prefetch bytes (scalar plus vector).
PRFB (vector plus immediate): Gather prefetch bytes (vector plus immediate).
PRFD (scalar plus immediate): Contiguous prefetch doublewords (immediate index).
PRFD (scalar plus scalar): Contiguous prefetch doublewords (scalar index).
PRFD (scalar plus vector): Gather prefetch doublewords (scalar plus vector).
PRFD (vector plus immediate): Gather prefetch doublewords (vector plus immediate).
PRFH (scalar plus immediate): Contiguous prefetch halfwords (immediate index).
PRFH (scalar plus scalar): Contiguous prefetch halfwords (scalar index).
PRFH (scalar plus vector): Gather prefetch halfwords (scalar plus vector).
PRFH (vector plus immediate): Gather prefetch halfwords (vector plus immediate).
PRFW (scalar plus immediate): Contiguous prefetch words (immediate index).
PRFW (scalar plus scalar): Contiguous prefetch words (scalar index).
PRFW (scalar plus vector): Gather prefetch words (scalar plus vector).
PRFW (vector plus immediate): Gather prefetch words (vector plus immediate).

PTEST: Set condition flags for predicate.

PTRUE, PTRUES: Initialise predicate from named constraint.
PUNPKHI, PUNPKLO: Unpack and widen half of predicate.
RADDHB: Rounding add narrow high part (bottom).
RADDHT: Rounding add narrow high part (top).
RAX1: Bitwise rotate left by 1 and exclusive OR.
RBIT: Reverse bits (predicated).

RDFR (unpredicated): Read the first-fault register.
RDFR, RDFRS (predicated): Return predicate of succesfully loaded elements.
RDVL: Read multiple of vector register size to scalar register.
REV (predicate): Reverse all elements in a predicate.
REV (vector): Reverse all elements in a vector (unpredicated).
REVB, REVH, REVW: Reverse bytes / halfwords / words within elements (predicated).
RSHRN: Rounding shift right narrow by immediate (bottom).
RSHRNT: Rounding shift right narrow by immediate (top).
RSUBNH: Rounding subtract narrow high part (bottom).
RSUBNHT: Rounding subtract narrow high part (top).
SABA: Signed absolute difference and accumulate.
SABALB: Signed absolute difference and accumulate long (bottom).
SABALT: Signed absolute difference and accumulate long (top).
**SABD**: Signed absolute difference (predicated).

**SABDLB**: Signed absolute difference long (bottom).

**SABDLT**: Signed absolute difference long (top).

**SADALP**: Signed add and accumulate long pairwise.

**SADDLB**: Signed add long (bottom).

**SADDLBT**: Signed add long (bottom + top).

**SADDLT**: Signed add long (top).

**SADDY**: Signed add reduction to scalar.

**SADDWB**: Signed add wide (bottom).

**SADDWT**: Signed add wide (top).

**SBCLB**: Subtract with carry long (bottom).

**SBCLT**: Subtract with carry long (top).

**SCVTF**: Signed integer convert to floating-point (predicated).

**SDIV**: Signed divide (predicated).

**SDIVR**: Signed reversed divide (predicated).

**SDOT (indexed)**: Signed integer indexed dot product.

**SDOT (vectors)**: Signed integer dot product.

**SEL (predicates)**: Conditionally select elements from two predicates.

**SEL (vectors)**: Conditionally select elements from two vectors.

**SETFFR**: Initialise the first-fault register to all true.

**SHADD**: Signed halving addition.

**SHRNB**: Shift right narrow by immediate (bottom).

**SHRNT**: Shift right narrow by immediate (top).

**SHSUB**: Signed halving subtract.

**SHSUBR**: Signed halving subtract reversed vectors.

**SLI**: Shift left and insert (immediate).

**SM4E**: SM4 encryption and decryption.

**SM4EKEY**: SM4 key updates.

**SMAX (immediate)**: Signed maximum with immediate (unpredicated).

**SMAX (vectors)**: Signed maximum vectors (predicated).

**SMAXP**: Signed maximum pairwise.

**SMAXV**: Signed maximum reduction to scalar.

**SMIN (immediate)**: Signed minimum with immediate (unpredicated).

**SMIN (vectors)**: Signed minimum vectors (predicated).

**SMINP**: Signed minimum pairwise.

**SMINV**: Signed minimum reduction to scalar.
SMLALB (indexed): Signed multiply-add long to accumulator (bottom, indexed).
SMLALB (vectors): Signed multiply-add long to accumulator (bottom).
SMLALT (indexed): Signed multiply-add long to accumulator (top, indexed).
SMLALT (vectors): Signed multiply-add long to accumulator (top).
SMLSLB (indexed): Signed multiply-subtract long from accumulator (bottom, indexed).
SMLSLB (vectors): Signed multiply-subtract long from accumulator (bottom).
SMLSLT (indexed): Signed multiply-subtract long from accumulator (top, indexed).
SMLSLT (vectors): Signed multiply-subtract long from accumulator (top).
SMMLA: Signed integer matrix multiply-accumulate.
SMULH (predicated): Signed multiply returning high half (predicated).
SMULH (unpredicated): Signed multiply returning high half (unpredicated).
SMULLB (indexed): Signed multiply long (bottom, indexed).
SMULLB (vectors): Signed multiply long (bottom).
SMULLT (indexed): Signed multiply long (top, indexed).
SMULLT (vectors): Signed multiply long (top).
SPLICE: Splice two vectors under predicate control.
SQABS: Signed saturating absolute value.
SQADD (immediate): Signed saturating add immediate (unpredicated).
SQADD (vectors, predicated): Signed saturating addition (predicated).
SQADD (vectors, unpredicated): Signed saturating add vectors (unpredicated).
SQCADD: Saturating complex integer add with rotate.
SQDECB: Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count.
SQDECD (scalar): Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count.
SQDECD (vector): Signed saturating decrement vector by multiple of 64-bit predicate constraint element count.
SQDECH (scalar): Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count.
SQDECH (vector): Signed saturating decrement vector by multiple of 16-bit predicate constraint element count.
SQDECP (scalar): Signed saturating decrement scalar by count of true predicate elements.
SQDECP (vector): Signed saturating decrement vector by count of true predicate elements.
SODMLALB (indexed): Signed saturating doubling multiply-add long to accumulator (bottom, indexed).
SODMLALB (vectors): Signed saturating doubling multiply-add long to accumulator (bottom).
SODMLALBT: Signed saturating doubling multiply-add long to accumulator (bottom × top).
SODMLALT (indexed): Signed saturating doubling multiply-add long to accumulator (top, indexed).
SODMLALT (vectors): Signed saturating doubling multiply-add long to accumulator (top).
SODMLSLLB (indexed): Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed).
**SODMLSLB (vectors)**: Signed saturating doubling multiply-subtract long from accumulator (bottom).

**SODMLSLBT**: Signed saturating doubling multiply-subtract long from accumulator (bottom × top).

**SODMLSLT (indexed)**: Signed saturating doubling multiply-subtract long from accumulator (top, indexed).

**SODMLSLT (vectors)**: Signed saturating doubling multiply-subtract long from accumulator (top).

**SODMULH (indexed)**: Signed saturating doubling multiply high (indexed).

**SODMULH (vectors)**: Signed saturating doubling multiply high (unpredicated).

**SODMULLB (indexed)**: Signed saturating doubling multiply long (bottom, indexed).

**SODMULLB (vectors)**: Signed saturating doubling multiply long (bottom).

**SODMULLT (indexed)**: Signed saturating doubling multiply long (top, indexed).

**SODMULLT (vectors)**: Signed saturating doubling multiply long (top).

**SQINCB**: Signed saturating increment scalar by multiple of 8-bit predicate constraint element count.

**SQINCD (scalar)**: Signed saturating increment scalar by multiple of 64-bit predicate constraint element count.

**SQINCD (vector)**: Signed saturating increment vector by multiple of 64-bit predicate constraint element count.

**SQINCH (scalar)**: Signed saturating increment scalar by multiple of 16-bit predicate constraint element count.

**SQINCH (vector)**: Signed saturating increment vector by multiple of 16-bit predicate constraint element count.

**SQINCP (scalar)**: Signed saturating increment scalar by count of true predicate elements.

**SQINCP (vector)**: Signed saturating increment vector by count of true predicate elements.

**SQINCW (scalar)**: Signed saturating increment scalar by multiple of 32-bit predicate constraint element count.

**SQINCW (vector)**: Signed saturating increment vector by multiple of 32-bit predicate constraint element count.

**SQNEG**: Signed saturating negate.

**SQRDCMLAH (indexed)**: Saturating rounding doubling complex integer multiply-add high with rotate (indexed).

**SQRDCMLAH (vectors)**: Saturating rounding doubling complex integer multiply-add high with rotate.

**SQRDMMLAH (indexed)**: Signed saturating rounding doubling multiply-add high to accumulator (indexed).

**SQRDMMLAH (vectors)**: Signed saturating rounding doubling multiply-add high to accumulator (unpredicated).

**SQRDMLSH (indexed)**: Signed saturating rounding doubling multiply-subtract high from accumulator (indexed).

**SQRDMLSH (vectors)**: Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated).

**SQRDMULH (indexed)**: Signed saturating rounding doubling multiply high (indexed).

**SQRDMULH (vectors)**: Signed saturating rounding doubling multiply high (unpredicated).

**SQRSHL**: Signed saturating rounding shift left by vector (predicated).

**SQRSHLR**: Signed saturating rounding shift left reversed vectors (predicated).

**SQRSHRNBP**: Signed saturating rounding shift right narrow by immediate (bottom).

**SQRSHRNT**: Signed saturating rounding shift right narrow by immediate (top).

**SQRSHRUNBP**: Signed saturating rounding shift right unsigned narrow by immediate (bottom).

**SQRSHRUNT**: Signed saturating rounding shift right unsigned narrow by immediate (top).

**SQSHL (immediate)**: Signed saturating shift left by immediate.

**SQSHL (vectors)**: Signed saturating shift left by vector (predicated).
SOSHLR: Signed saturating shift left reversed vectors (predicated).
SOSHLU: Signed saturating shift left unsigned by immediate.
SOSHRNB: Signed saturating shift right narrow by immediate (bottom).
SOSHRNT: Signed saturating shift right narrow by immediate (top).
SOSHRUNB: Signed saturating shift right unsigned narrow by immediate (bottom).
SOSHRUNT: Signed saturating shift right unsigned narrow by immediate (top).
SOSUB (immediate): Signed saturating subtract immediate (unpredicated).
SOSUB (vectors, predicated): Signed saturating subtraction (predicated).
SOSUB (vectors, unpredicated): Signed saturating subtract vectors (unpredicated).
SOSUBR: Signed saturating subtraction reversed vectors (predicated).
SOXTNB: Signed saturating extract narrow (bottom).
SOXTNT: Signed saturating extract narrow (top).
SOXTUNB: Signed saturating unsigned extract narrow (bottom).
SOXTUNT: Signed saturating unsigned extract narrow (top).
SRHADD: Signed rounding halving addition.
SRI: Shift right and insert (immediate).
SRSHL: Signed rounding shift left by vector (predicated).
SRSHLR: Signed rounding shift left reversed vectors (predicated).
SRSHR: Signed rounding shift right by immediate.
SRSRA: Signed rounding shift right and accumulate (immediate).
SSHLLB: Signed shift left long by immediate (bottom).
SSHLLT: Signed shift left long by immediate (top).
SSRA: Signed shift right and accumulate (immediate).
SSUBLB: Signed subtract long (bottom).
SSUBLBT: Signed subtract long (bottom - top).
SSUBLT: Signed subtract long (top).
SSUBLTB: Signed subtract long (top - bottom).
SSUBWB: Signed subtract wide (bottom).
SSUBWT: Signed subtract wide (top).
ST1B (scalar plus immediate): Contiguous store bytes from vector (immediate index).
ST1B (scalar plus scalar): Contiguous store bytes from vector (scalar index).
ST1B (scalar plus vector): Scatter store bytes from a vector (vector index).
ST1B (vector plus immediate): Scatter store bytes from a vector (immediate index).
ST1D (scalar plus immediate): Contiguous store doublewords from vector (immediate index).
ST1D (scalar plus scalar): Contiguous store doublewords from vector (scalar index).
ST1D (scalar plus vector): Scatter store doublewords from a vector (vector index).
ST1D (vector plus immediate): Scatter store doublewords from a vector (immediate index).
ST1H (scalar plus immediate): Contiguous store halfwords from vector (immediate index).
ST1H (scalar plus scalar): Contiguous store halfwords from vector (scalar index).
ST1H (vector plus immediate): Scatter store halfwords from a vector (vector index).
ST1H (vector plus immediate): Scatter store halfwords from a vector (immediate index).
ST1W (scalar plus immediate): Contiguous store words from vector (immediate index).
ST1W (scalar plus scalar): Contiguous store words from vector (scalar index).
ST1W (scalar plus vector): Scatter store words from a vector (vector index).
ST1W (vector plus immediate): Scatter store words from a vector (immediate index).
ST2B (scalar plus immediate): Contiguous store two-byte structures from two vectors (immediate index).
ST2B (scalar plus scalar): Contiguous store two-byte structures from two vectors (scalar index).
ST2D (scalar plus immediate): Contiguous store two-doubleword structures from two vectors (immediate index).
ST2D (scalar plus scalar): Contiguous store two-doubleword structures from two vectors (scalar index).
ST2H (scalar plus immediate): Contiguous store two-halfword structures from two vectors (immediate index).
ST2H (scalar plus scalar): Contiguous store two-halfword structures from two vectors (scalar index).
ST2W (scalar plus immediate): Contiguous store two-word structures from two vectors (immediate index).
ST2W (scalar plus scalar): Contiguous store two-word structures from two vectors (scalar index).
ST3B (scalar plus immediate): Contiguous store three-byte structures from three vectors (immediate index).
ST3B (scalar plus scalar): Contiguous store three-byte structures from three vectors (scalar index).
ST3D (scalar plus immediate): Contiguous store three-doubleword structures from three vectors (immediate index).
ST3D (scalar plus scalar): Contiguous store three-doubleword structures from three vectors (scalar index).
ST3H (scalar plus immediate): Contiguous store three-halfword structures from three vectors (immediate index).
ST3H (scalar plus scalar): Contiguous store three-halfword structures from three vectors (scalar index).
ST3W (scalar plus immediate): Contiguous store three-word structures from three vectors (immediate index).
ST3W (scalar plus scalar): Contiguous store three-word structures from three vectors (scalar index).
ST4B (scalar plus immediate): Contiguous store four-byte structures from four vectors (immediate index).
ST4B (scalar plus scalar): Contiguous store four-byte structures from four vectors (scalar index).
ST4D (scalar plus immediate): Contiguous store four-doubleword structures from four vectors (immediate index).
ST4D (scalar plus scalar): Contiguous store four-doubleword structures from four vectors (scalar index).
ST4H (scalar plus immediate): Contiguous store four-halfword structures from four vectors (immediate index).
ST4H (scalar plus scalar): Contiguous store four-halfword structures from four vectors (scalar index).
ST4W (scalar plus immediate): Contiguous store four-word structures from four vectors (immediate index).
ST4W (scalar plus scalar): Contiguous store four-word structures from four vectors (scalar index).
STNT1B (scalar plus immediate): Contiguous store non-temporal bytes from vector (immediate index).
STNT1B (scalar plus scalar): Contiguous store non-temporal bytes from vector (scalar index).
STNT1B (vector plus scalar): Scatter store non-temporal bytes.
STNT1D (scalar plus immediate): Contiguous store non-temporal doublewords from vector (immediate index).
STNT1D (scalar plus scalar): Contiguous store non-temporal doublewords from vector (scalar index).
STNT1D (vector plus scalar): Scatter store non-temporal doublewords.
STNT1H (scalar plus immediate): Contiguous store non-temporal halfwords from vector (immediate index).
STNT1H (scalar plus scalar): Contiguous store non-temporal halfwords from vector (scalar index).
STNT1H (vector plus scalar): Scatter store non-temporal halfwords.
STNT1W (scalar plus immediate): Contiguous store non-temporal words from vector (immediate index).
STNT1W (scalar plus scalar): Contiguous store non-temporal words from vector (scalar index).
STNT1W (vector plus scalar): Scatter store non-temporal words.
STR (predicate): Store predicate register.
STR (vector): Store vector register.
SUB (immediate): Subtract immediate (unpredicated).
SUB (vectors, predicated): Subtract vectors (predicated).
SUB (vectors, unpredicated): Subtract vectors (unpredicated).
SUBHNB: Subtract narrow high part (bottom).
SUBHNT: Subtract narrow high part (top).
SUBR (immediate): Reversed subtract from immediate (unpredicated).
SUBR (vectors): Reversed subtract vectors (predicated).
SUDOT: Signed by unsigned integer indexed dot product.
SUNPKHI, SUNPKLO: Signed unpack and extend half of vector.
SUQADD: Signed saturating addition of unsigned value.
SXTB, SXTH, SXTW: Signed byte / halfword / word extend (predicated).
TBL: Programmable table lookup in one or two vector table (zeroing).
TBX: Programmable table lookup in single vector table (merging).
TRN1, TRN2 (predicates): Interleave even or odd elements from two predicates.
TRN1, TRN2 (vectors): Interleave even or odd elements from two vectors.
UABA: Unsigned absolute difference and accumulate.
UABALB: Unsigned absolute difference and accumulate long (bottom).
UABALT: Unsigned absolute difference and accumulate long (top).
UABD: Unsigned absolute difference (predicated).
UABD LB: Unsigned absolute difference long (bottom).
UABDLT: Unsigned absolute difference long (top).
UADALP: Unsigned add and accumulate long pairwise.
UADDLB: Unsigned add long (bottom).
UADDLT: Unsigned add long (top).
UADDV: Unsigned add reduction to scalar.
**UADDWP**: Unsigned add wide (bottom).

**UADDWT**: Unsigned add wide (top).

**UCVTF**: Unsigned integer convert to floating-point (predicated).

**UDIV**: Unsigned divide (predicated).

**UDIVR**: Unsigned reversed divide (predicated).

**UDOT (indexed)**: Unsigned integer indexed dot product.

**UDOT (vectors)**: Unsigned integer dot product.

**UHADD**: Unsigned halving addition.

**UHSUB**: Unsigned halving subtract.

**UHSUBR**: Unsigned halving subtract reversed vectors.

**UMAX (immediate)**: Unsigned maximum with immediate (unpredicated).

**UMAX (vectors)**: Unsigned maximum vectors (predicated).

**UMAXP**: Unsigned maximum pairwise.

**UMAXV**: Unsigned maximum reduction to scalar.

**UMIN (immediate)**: Unsigned minimum with immediate (unpredicated).

**UMIN (vectors)**: Unsigned minimum vectors (predicated).

**UMINP**: Unsigned minimum pairwise.

**UMINY**: Unsigned minimum reduction to scalar.

**UMLALB (indexed)**: Unsigned multiply-add long to accumulator (bottom, indexed).

**UMLALB (vectors)**: Unsigned multiply-add long to accumulator (bottom).

**UMLALT (indexed)**: Unsigned multiply-add long to accumulator (top, indexed).

**UMLALT (vectors)**: Unsigned multiply-add long to accumulator (top).

**UMLSLB (indexed)**: Unsigned multiply-subtract long from accumulator (bottom, indexed).

**UMLSLB (vectors)**: Unsigned multiply-subtract long from accumulator (bottom).

**UMLSLT (indexed)**: Unsigned multiply-subtract long from accumulator (top, indexed).

**UMLSLT (vectors)**: Unsigned multiply-subtract long from accumulator (top).

**UMMLA**: Unsigned integer matrix multiply-accumulate.

**UMULH (predicated)**: Unsigned multiply returning high half (predicated).

**UMULH (unpredicated)**: Unsigned multiply returning high half (unpredicated).

**UMULLB (indexed)**: Unsigned multiply long (bottom, indexed).

**UMULLB (vectors)**: Unsigned multiply long (bottom).

**UMULLT (indexed)**: Unsigned multiply long (top, indexed).

**UMULLT (vectors)**: Unsigned multiply long (top).

**UQADD (immediate)**: Unsigned saturating add immediate (unpredicated).

**UQADD (vectors, predicated)**: Unsigned saturating addition (predicated).

**UQADD (vectors, unpredicated)**: Unsigned saturating add vectors (unpredicated).
A64 -- SVE Instructions (alphabetic order)

**UQDECB**: Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count.

**UQDECD (scalar)**: Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count.

**UQDECD (vector)**: Signed saturating decrement vector by multiple of 64-bit predicate constraint element count.

**UQDECH (scalar)**: Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count.

**UQDECH (vector)**: Signed saturating decrement vector by multiple of 16-bit predicate constraint element count.

**UQDECP (scalar)**: Signed saturating decrement scalar by count of true predicate elements.

**UQDECP (vector)**: Signed saturating decrement vector by count of true predicate elements.

**UQDECW (scalar)**: Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.

**UQDECW (vector)**: Signed saturating decrement vector by multiple of 32-bit predicate constraint element count.

**UQINCB**: Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count.

**UQINCD (scalar)**: Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count.

**UQINCD (vector)**: Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count.

**UQINCH (scalar)**: Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count.

**UQINCH (vector)**: Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count.

**UQINCP (scalar)**: Unsigned saturating increment scalar by count of true predicate elements.

**UQINCP (vector)**: Unsigned saturating increment vector by count of true predicate elements.

**UQINCW (scalar)**: Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count.

**UQINCW (vector)**: Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count.

**UQRSHL**: Unsigned saturating rounding shift left by vector (predicated).

**UQRSHLR**: Unsigned saturating rounding shift left reversed vectors (predicated).

**UQSHLRNB**: Unsigned saturating rounding shift right narrow by immediate (bottom).

**UQSHRNT**: Unsigned saturating rounding shift right narrow by immediate (top).

**UQSHL (immediate)**: Unsigned saturating shift left by immediate.

**UQSHL (vectors)**: Unsigned saturating shift left by vector (predicated).

**UQSHLR**: Unsigned saturating shift left reversed vectors (predicated).

**UQSHRN**: Unsigned saturating shift right narrow by immediate (bottom).

**UQSHRT**: Unsigned saturating shift right narrow by immediate (top).

**UQSUB (immediate)**: Unsigned saturating subtract immediate (unpredicated).

**UQSUB (vectors, predicated)**: Unsigned saturating subtraction (predicated).

**UQSUB (vectors, unpredicated)**: Unsigned saturating subtract vectors (unpredicated).

**UQSUBR**: Unsigned saturating subtraction reversed vectors (predicated).

**UQXTNB**: Unsigned saturating extract narrow (bottom).

**UQXTNT**: Unsigned saturating extract narrow (top).

**URECPE**: Unsigned reciprocal estimate (predicated).

**URHADD**: Unsigned rounding halving addition.

**URSHL**: Unsigned rounding shift left by vector (predicated).
URSHLR: Unsigned rounding shift left reversed vectors (predicated).
URSHR: Unsigned rounding shift right by immediate.
URSORTE: Unsigned reciprocal square root estimate (predicated).
URSRA: Unsigned rounding shift right and accumulate (immediate).
USDOT (indexed): Unsigned by signed integer indexed dot product.
USDOT (vectors): Unsigned by signed integer dot product.
USHLLB: Unsigned shift left long by immediate (bottom).
USHLLT: Unsigned shift left long by immediate (top).
USMMLA: Unsigned by signed integer matrix multiply-accumulate.
USQADD: Unsigned saturating addition of signed value.
USRA: Unsigned shift right and accumulate (immediate).
USUBLB: Unsigned subtract long (bottom).
USUBLT: Unsigned subtract long (top).
USUBWB: Unsigned subtract wide (bottom).
USUBWT: Unsigned subtract wide (top).
UUNPKHI, UUNPKLO: Unsigned unpack and extend half of vector.
UXTB, UXTH, UXTW: Unsigned byte / halfword / word extend (predicated).
UZP1, UZP2 (predicates): Concatenate even or odd elements from two predicates.
UZP1, UZP2 (vectors): Concatenate even or odd elements from two vectors.
WHILEGE: While decrementing signed scalar greater than or equal to scalar.
WHILEGT: While decrementing signed scalar greater than scalar.
WHILEHI: While decrementing unsigned scalar higher than scalar.
WHILEHS: While decrementing unsigned scalar higher or same as scalar.
WHILELE: While incrementing signed scalar less than or equal to scalar.
WHILELO: While incrementing unsigned scalar lower than scalar.
WHILELS: While incrementing unsigned scalar lower or same as scalar.
WHILELT: While incrementing unsigned scalar less than scalar.
WHILERW: While free of read-after-write conflicts.
WHILEWR: While free of write-after-read/write conflicts.
WRFR: Write the first-fault register.
XAR: Bitwise exclusive OR and rotate right by immediate.
ZIP1, ZIP2 (predicates): Interleave elements from two half predicates.
ZIP1, ZIP2 (vectors): Interleave elements from two half vectors.
ABS

Absolute value (predicated).

Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

ABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
  integer element = SInt(Elem[operand, e, esize]);
  if ElemP[mask, e, esize] == '1' then
    element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

  Z[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
The **MOVPRFX** instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

The **MOVPRFX** instruction must specify the same destination register as this instruction.

The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ADCLB

Add with carry long (bottom).

Add the even-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

![Binary Representation]

ADCLB <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<T> Is the size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n];
bits(VL) carries = Z[m];
bits(VL) result = Z[da];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 0, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;
    (res, nzcv) = AddWithCarry(element1, element2, carry_in);
    carry_out = nzcv<1>;
    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out);

Z[da] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ADCLT

Add with carry long (top).

Add the odd-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 | 0 1 0 | sz | 0   |   | Zm   | 1 1 0 1 0 1 0 | 1   | Zn   | Zda T
```

ADCLT <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<T> Is the size specifier, encoded in “sz”:

```
<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n];
bits(VL) carries = Z[m];
bits(VL) result = Z[da];
for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 1, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;
    (res, nzcv) = AddWithCarry(element1, element2, carry_in);
    carry_out = nzcv<1>;
    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out);
Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**ADD (vectors, predicated)**

Add vectors (predicated).

Add active elements of the second source vector to corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 0 | size | 0 0 0 0 0 0 0 | Pg | Zm | Zdn |

ADD <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>,<T>

if !HaveSVE() then UNDEFINED;
ninteger esize = 8 << UInt(size);
ninteger g = UInt(Pg);
ninteger dn = UInt(Zdn);
ninteger m = UInt(Zm);

**Assembler Symbols**

&lt;Zdn&gt;  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
&lt;T&gt;  Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

&lt;Pg&gt;  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
&lt;Zm&gt;  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
ninteger elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1

bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
if ElemP[mask, e, esize] == '1' then

Elem[result, e, esize] = element1 + element2;
else

Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADD (immediate)

Add immediate (unpredicated).

Add an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 + imm;
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ADD (vectors, unpredicated)

Add vectors (unpredicated).

Add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|    |
|   | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |    |
| Zm|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Zn|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Zd|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

ADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 + element2;
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADDHNB

Add narrow high part (bottom).

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 | size | 1 | Zm | 0 1 1 0 0 0 0 | Zn | Zd
S R T
```

ADDHNB <Zd>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer halfesize = esize DIV 2;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 + element2) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros();
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ADDHNT

Add narrow high part (top).

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 | size | 1 | Zm | 0 1 1 0 0 1 | Zn | Zd
S R T

ADDHNT <Zd>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[d];
integer halfsize = esize DIV 2;

for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  integer res = (element1 + element2) >> halfsize;
  Elem[result, 2*e + 1, halfsize] = res<halfsize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADDP

Add pairwise.

Add pairs of adjacent elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|----------------|
| 0 1 0 0 0 1 0 0 | 0 1 0 0 0 1 1 0 | P g | Z m | Z d n |

ADDP <Z d n>..<T>, <P g>..M, <Z d n>..<T>, <Z m>..<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Z d n> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<P g> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Z m> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = _VL_ DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = UInt(Elem[operand1, e + 0, esize]);
      element2 = UInt(Elem[operand1, e + 1, esize]);
    else
      element1 = UInt(Elem[operand2, e - 1, esize]);
      element2 = UInt(Elem[operand2, e + 0, esize]);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ADDPL

Add multiple of predicate register size to scalar register.

Add the current predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | Rn | 0  | 1  | 0  | 1  | 0  | imm6| Rd |

ADDPL <Xd|SP>, <Xn|SP>, #<imm>

if !HaveSVE() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler Symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSVEEnabled();

bits(64) operand1 = if n == 31 then SP[] else X[n];

bits(64) result = operand1 + (imm * (PL DIV 8));

if d == 31 then
    SP[] = result;
else
    X[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
      ◦ The values of the data supplied in any of its registers.
      ◦ The values of the NZCV flags.
ADDVL

Add multiple of vector register size to scalar register.

Add the current vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.

|   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0 |
|---------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
|      0 0 0 0 0 1 0 0 0 1 1      |   Rn   | 0 1 0 1 0       | imm6          |   Rd   |

ADDVL <Xd|SP>, <Xn|SP>, #<imm>

if !HaveSVE() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);

Assembler Symbols

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP> Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
bits(64) operand1 = if n == 31 then SP[] else X[n];
bits(64) result = operand1 + (imm * (VL DIV 8));
if d == 31 then
    SP[] = result;
else
    X[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ADR

Compute vector address.

Optionally sign or zero-extend the least significant 32-bits of each element from a vector of offsets or indices in the second source vector, scale each index by 2, 4 or 8, add to a vector of base addresses from the first source vector, and place the resulting addresses in the destination vector. This instruction is unpredicated.

It has encodings from 3 classes: Packed offsets, Unpacked 32-bit signed offsets and Unpacked 32-bit unsigned offsets

Packed offsets

If !HaveSVE() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer osize = esize;
boolean unsigned = TRUE;
integer mbytes = 1 << UInt(msz);

Unpacked 32-bit signed offsets

If !HaveSVE() then UNDEFINED;
integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer osize = 32;
boolean unsigned = FALSE;
integer mbytes = 1 << UInt(msz);

Unpacked 32-bit unsigned offsets

If !HaveSVE() then UNDEFINED;
integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer osize = 32;
boolean unsigned = TRUE;
integer mbytes = 1 << UInt(msz);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "sz":

ADR
<sz> <T>
0  S
1  D

<Zn> Is the name of the base scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the offset scalable vector register, encoded in the “Zm” field.

<mod> Is the index extend and shift specifier, encoded in “msz”:

<table>
<thead>
<tr>
<th>msz</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>[absent]</td>
</tr>
<tr>
<td>x1</td>
<td>LSL</td>
</tr>
<tr>
<td>10</td>
<td>LSL</td>
</tr>
</tbody>
</table>

<amount> Is the index shift amount, encoded in “msz”:

<table>
<thead>
<tr>
<th>msz</th>
<th>&lt;amount&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>[absent]</td>
</tr>
<tr>
<td>01</td>
<td>#1</td>
</tr>
<tr>
<td>10</td>
<td>#2</td>
</tr>
<tr>
<td>11</td>
<td>#3</td>
</tr>
</tbody>
</table>

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) offs = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) addr = Elem[base, e, esize];
    integer offset = Int(Elem[offs, e, esize]<osize-1:0>, unsigned);
    Elem[result, e, esize] = addr + (offset * mbytes);
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESD

AES single round decryption.

The AESD instruction reads a 16-byte state array from each 128-bit segment of the first source vector, together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the ADDROUNDKEY(), INVSUBBYTES() and INVSHIFTROWS() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   1   0   0   1   0   1   0   0   1   0   1   1   1   0   0   1   Zm   | Zdn   |
```

size<1>size<0>


if !HaveSVE2AES() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
result = operand1 EOR operand2;
for s = 0 to segments-1
    Elem[result, s, 128] = AESInvSubBytes(AESInvShiftRows(Elem[result, s, 128]));
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AESE

AES single round encryption.

The AESE instruction reads a 16-byte state array from each 128-bit segment of the first source vector together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the ADDROUNDKEY(), SUBBYTES() and SHIFTROWS() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 1 0 0 0 1 0 1 0 | 0 1 0 0 0 1 0 1 0 | 1 1 1 0 0 0 0 0 0 |


if !HaveSVE2AES() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
result = operand1 EOR operand2;
for s = 0 to segments-1
    Elem[result, s, 128] = AESSubBytes(AESShiftRows(Elem[result, s, 128]));
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESIMC

AES inverse mix columns.

The AESIMC instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the INVMIXCOLUMNS() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand = Z[dn];
bits(VL) result;
for s = 0 to segments-1
    Elem[result, s, 128] = AESInvMixColumns(Elem[operand, s, 128]);
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AESMC

AES mix columns.

The AESMC instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the MIXCOLUMNS() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

AESMC <Zdn>.B, <Zdn>.B

if !HaveSVE2AES() then UNDEFINED;
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand = Z[dn];
bits(VL) result;

for s = 0 to segments-1
    Elem[result, s, 128] = AESMixColumns(Elem[operand, s, 128]);

Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
AND, ANDS (predicates)

Bitwise AND predicates.

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the aliases MOV (predicated), and MOV (predicate, predicated, zeroing).

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

S


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

S


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (predicated)</td>
<td>( S == '1' &amp;&amp; \text{Pn} == \text{Pm} )</td>
</tr>
<tr>
<td>MOV (predicate, predicated, zeroing)</td>
<td>( S == '0' &amp;&amp; \text{Pn} == \text{Pm} )</td>
</tr>
</tbody>
</table>
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
AND (vectors, predicated)

Bitwise AND vectors (predicated).

Bitwise AND active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
           size   Pg   Zm   Zdn
0 0 0 0 0 1 0 0
```

AND <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>,<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

```
size  <T>
00 B
01 H
10 S
11 D
```

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 AND element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
AND (immediate)

Bitwise AND with immediate (unpredicated).

Bitwise AND an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction **BIC (immediate)**.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

- **<Zdn>** is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<const>** is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

**Operation**

```assembly
define CheckSVEEnabled; integer elements = VL DIV 64; bits(VL) operand = Z[dn]; bits(VL) result;
for e = 0 to elements-1
    bits(64) elements1 = Elem[operand, e, 64];
    Elem[result, e, 64] = elements1 AND imm;
Z[dn] = result;```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
AND (vectors, unpredicated)

Bitwise AND vectors (unpredicated).

Bitwise AND all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | Zm | 0  | 0  | 1  | 1  | 0  | 0  | 1  | Zn | 0  | 0  | Zd |

AND <Zd>.D, <Zn>.D, <Zm>.D

if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
Z[d] = operand1 AND operand2;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ANDV

Bitwise AND reduction to scalar.

Bitwise AND horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as all ones.

\[
\begin{array}{cccccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & Pg & Zn & Vd
\end{array}
\]

\[
\text{ANDV } <V>, <d>, <Pg>, <Zn>.<T>
\]

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer n = Uint(Zn);
integer d = Uint(Vd);

Assembler Symbols

\(<V>\) 
Is a width specifier, encoded in “size”:
\[
\begin{array}{c|c}
\text{size} & <V> \\
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\end{array}
\]

\(<d>\) 
Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

\(<Pg>\) 
Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

\(<Zn>\) 
Is the name of the source scalable vector register, encoded in the “Zn” field.

\(<T>\) 
Is the size specifier, encoded in “size”:
\[
\begin{array}{c|c}
\text{size} & <T> \\
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\end{array}
\]

Operation

\(\text{CheckSVEEnabled();}\)
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) result = Ones(esize);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    result = result AND Elem[operand, e, esize];
V[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
The values of the NZCV flags.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ASR (immediate, predicated)**

Arithmetic shift right by immediate (predicated).

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ASR <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, #<const>**

\[
\text{if } !\text{HaveSVE}() \text{ then UNDEFINED;}
\]

\[
\text{bits}(4) \text{ tsize = tszh:tszl;}
\]

\[
\text{case tsize of}
\]\n
\[
\begin{align*}
\text{when '0000' UNDEFINED;} \\
\text{when '0001' esize = 8;} \\
\text{when '001x' esize = 16;} \\
\text{when '01xx' esize = 32;} \\
\text{when '1xxx' esize = 64;}
\end{align*}
\]

\[
\text{integer g = UInt(Pg);} \\
\text{integer dn = UInt(Zdn);} \\
\text{integer shift = (2 * esize) - UInt(tsize:imm3);} \\
\]

**Assembler Symbols**

- **<Zdn>**
  - Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>**
  - Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>**
  - Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<const>**
  - Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

\[
\text{CheckSVEEnabled();}
\]

\[
\text{integer elements = VL DIV esize;}
\]

\[
\text{bits(VL) operand1 = Z[dn];}
\]

\[
\text{bits(PL) mask = P[g];}
\]

\[
\text{bits(VL) result;}
\]

\[
\text{for } e = 0 \text{ to elements-1}
\]

\[
\text{bits(esize) element1 = Elem[operand1, e, esize];}
\]

\[
\text{if Elem[mask, e, esize] == '1' then}
\]

\[
\text{Elem[result, e, esize] = ASR(element1, shift);} \\
\]

\[
\text{else}
\]

\[
\text{Elem[result, e, esize] = Elem[operand1, e, esize];}
\]

\[
Z[dn] = \text{result;}
\]

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a **MOVPRFX** instruction. The **MOVPRFX** instruction must conform to all of the following requirements, otherwise the behavior of the **MOVPRFX** and this instruction is UNPREDICTABLE:

- The **MOVPRFX** instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The **MOVPRFX** instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
### ASR (wide elements, predicated)

Arithmetic shift right by 64-bit wide elements (predicated).

Shift right, preserving the sign bit, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0 1 1 0 0 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ASR** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

if **!HaveSVE()** then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

#### Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

#### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bite(PL) mask = P[g];
bite(VL) operand1 = Z[dn];
bite(VL) operand2 = Z[m];
bite(VL) result;
for e = 0 to elements-1
    bite(64) element1 = Elem[operand1, e, esize];
bite(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    if ElemP[mask, e, esize] == '1' then 
        Elem[result, e, esize] = ASR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

#### Operational information

If PSTATE.DIT is 1:

* The execution time of this instruction is independent of:
  * The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
• The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ASR (vectors)

Arithmetic shift right by vector (predicated).

Shift right, preserving the sign bit, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 0 0 0 0 1 0 0 size 0 1 0 0 0 0 1 0 0 Pg Zm Zdn

ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element2), esize);
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = ASR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**ASR (immediate, unpredicated)**

Arithmetic shift right by immediate (unpredicated).

Shift right by immediate, preserving the sign bit, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

ASR $<Zd>.<T>$, $<Zn>.<T>$, $<\text{const}>$

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;

integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

**Assembler Symbols**

$<Zd>$ Is the name of the destination scalable vector register, encoded in the "Zd" field.

$<T>$ Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>lx</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>lx</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

$<Zn>$ Is the name of the source scalable vector register, encoded in the "Zn" field.

$<\text{const}>$ Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = ASR(element1, shift);
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ASR (wide elements, unpredicated)

Arithmetic shift right by 64-bit wide elements (unpredicated).

Shift right, preserving the sign bit, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.

```
0 0 0 0 0 1 0 0
size
Zm
1 0 0 0 0 0 0
Zn
0 0
Zd
```

ASR <Zd>.<T>, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = ASR(element1, shift);
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ASRD**

Arithmetic shift right for divide by immediate (predicated).

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The result rounds toward zero as in a signed division. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 | tszh | 0 0 | 0 1 | 0 1 0 0 | Pg | tszl | imm3 | Zdn
```

**ASRD**  
`<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>`

```plaintext
if !HaveSVE() then UNDEFINED;
b_bits(4) tsize = tszh:tszl;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsz:imm3);
```

**Assembler Symbols**

- `<Zdn>` is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<const>` is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        if element1 < 0 then
            element1 = element1 + ((1 << shift) - 1);
        Elem[result, e, esize] = (element1 >> shift)<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ASRR

Reversed arithmetic shift right by vector (predicated).

Reversed shift right, preserving the sign bit, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0 | size 0 1 0 1 0 0 | Pg | Zm | Zdn |

ASRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = ASR(element2, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BCAX

Bitwise clear and exclusive OR.

Bitwise AND elements of the second source vector with the corresponding inverted elements of the third source vector, then exclusive OR the results with corresponding elements of the first source vector. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | Zm | 0  | 0  | 1  | 1  | 1  | 0  | Zk | Zdn|


if !HaveSVE2() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

CheckSVEEnabled();
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[k];
Z[dn] = operand1 EOR (operand2 AND NOT(operand3));

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BDEP

Scatter lower bits into positions selected by bitmask.

This instruction scatters the lowest-numbered contiguous bits within each element of the first source vector to the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector, preserving their order, and set the bits corresponding to a zero mask bit to zero. This instruction is unpredicated.

ID_AA64FVR0_EL1.BitPerm indicates whether this instruction is implemented.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>1 0 1 1 0 1</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

BDEP <Zd>,<T>, <Zn>,<T>, <Zm>,<T>

if !HaveSVE2BitPerm() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) data = Z[n];
bits(VL) mask = Z[m];
bits(VL) result;
for e = 0 to elements - 1
  Elem[result, e, esize] = BitDeposit(Elem[data, e, esize], Elem[mask, e, esize]);
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BEXT

Gather lower bits from positions selected by bitmask.

This instruction gathers bits in each element of the first source vector from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order, and sets the remaining higher-numbered bits to zero. This instruction is unpredicated.

ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.

Size: 8

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------|---------------|-----------------|
| 0 1 0 0 0 1 0 1 | size | 0 | Zm | 1 0 1 1 0 0 | Zn | Zd |

BEXT <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2BitPerm() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) data = Z[n];
bits(VL) mask = Z[m];
bits(VL) result;
for e = 0 to elements - 1
    Elem[result, e, esize] = BitExtract(Elem[data, e, esize], Elem[mask, e, esize]);
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BFCVT

Floating-point down convert to BFloat16 format (predicated).

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the result type is smaller than the input type, the results are zero-extended to fill each destination element. Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction honors all of the FPCR bits that apply to single-precision arithmetic. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

```
0 1 1 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Pg Zn Zd
```

BFCVT <Zd>.H, <Pg>/M, <Zn>.S

if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(32) element = Elem[operand, e, 32];
    if ElemP[mask, e, 32] == '1'
        Elem[result, 2*e, 16] = FPConvertBF(element, FPCR[]);
        Elem[result, 2*e+1, 16] = Zeros();

Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**BFCVTNT**

Floating-point down convert and narrow to BFloat16 (top, predicated).

Convert active 32-bit single-precision elements from the source vector to BFloat16 format, and place the results in the odd-numbered 16-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.

Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction honors all of the FPCR bits that apply to single-precision arithmetic. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 0 1 1 0 0 1 0 0 | 1 0 | 0 0 1 0 | 1 0 | 1 0 1            Pg   Zn   Zd

BFCVTNT <Zd>.H, <Pg>/M, <Zn>.S
```

if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.

**Operation**

- `CheckSVEEnabled();`
- integer elements = VL DIV 32;
- bits(PL) mask = P[g];
- bits(VL) operand = Z[n];
- bits(VL) result = Z[d];

for e = 0 to elements-1
-  bits(32) element = Elem[operand, e, 32];
-  if ElemP[mask, e, 32] == '1' then
-    Elem[result, 2*e+1, 16] = FPConvertBF(element, FPCR[]);

Z[d] = result;
**BFDOT (vectors)**

**BFloat16 floating-point dot product.**

The BFloat16 floating-point (BF16) dot product instruction computes the dot product of a pair of BF16 values held in each 32-bit element of the first source vector multiplied by a pair of BF16 values in the corresponding 32-bit element of the second source vector, and then destructively adds the single-precision dot product to the corresponding single-precision element of the destination vector.

This instruction is unpredicated.

All floating-point calculations performed by this instruction are performed with the following behaviors, irrespective of the value in FPCR:

- Uses the non-IEEE 754 Round-to-Odd mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- The cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC and IOC) are not modified.
- Trapped floating-point exceptions are disabled, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE and IOE) are all zero.
- Denormalized inputs and results are flushed to zero, as if FPCR.FZ == 1.
- Only the Default NaN is generated, as if FPCR.DN == 1.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

---

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
  bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
  bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
  bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
  bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];

  bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
  Elem[result, e, 32] = BFAdd(Elem[operand3, e, 32], sum);

Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**BFDOT (indexed)**

BFloat16 floating-point indexed dot product.

The BFloat16 floating-point (BF16) indexed dot product instruction computes the dot product of a pair of BF16 values held in each 32-bit element of the first source vector multiplied by a pair of BF16 values in an indexed 32-bit element of the second source vector, and then destructively adds the single-precision dot product to the corresponding single-precision element of the destination vector.

The BF16 pairs within the second source vector are specified using an immediate index which selects the same BF16 pair position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated. All floating-point calculations performed by this instruction are performed with the following behaviors, irrespective of the value in FPCR:
* Uses the non-IEEE 754 Round-to-Odd mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
* The cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC and IOC) are not modified.
* Trapped floating-point exceptions are disabled, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE and IOE) are all zero.
* Denormalized inputs and results are flushed to zero, as if FPCR.FZ == 1.
* Only the Default NaN is generated, as if FPCR.DN == 1.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

```assembly

if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i2);
```

**Assembler Symbols**

- `<Zda>`: Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>`: Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
- `<imm>`: Is the immediate index, in the range 0 to 3, encoded in the "i2" field.

**Operation**

```assembly
CheckSVEEnabled();
integer elements = VL DIV 32;
integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
    Elem[result, e, 32] = BFAdd(Elem[operand3, e, 32], sum);
Z[da] = result;
```

BFDOT (indexed)  Page 1522
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFMLALB (vectors)

BFLOAT16 floating-point multiply-add long to single-precision (bottom).

This BFLOAT16 floating-point multiply-add long instruction widens the even-numbered 16-bit BFLOAT16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

Unlike the BFLOAT16 matrix multiplication and dot product instructions, this instruction performs a fused multiply-add that honors all of the FPCR bits that apply to single-precision arithmetic. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
Zm 1 0 0 0 0 Zn 0 0 0 0 0 0
Zda
```


```plaintext
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  bits(32) element1 = Elem[operand1, 2 * e + 0, 16] : Zeros(16);
  bits(32) element2 = Elem[operand2, 2 * e + 0, 16] : Zeros(16);
  bits(32) element3 = Elem[operand3, e, 32];
  Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand or register of this instruction.
BFMLALB (indexed)

BFloat16 floating-point multiply-add long to single-precision (bottom, indexed).

This BFloat16 floating-point multiply-add long instruction widens the even-numbered 16-bit BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated. Unlike the BFloat16 matrix multiplication and dot product instructions, this instruction performs a fused multiply-add that honors all of the FPCR bits that apply to single-precision arithmetic. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 0 1 1 1 i3h  Zm 0 1 0 0 i3l 0  Zn  Zda


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
<iimm> Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

CheckSVEEnabled();
integer elements = VL DIV 32;
integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(32) element1 = Elem[operand1, 2 * e + 0, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, s, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFMLALT (vectors)

BFLOAT16 floating-point multiply-add long to single-precision (top).

This BFLOAT16 floating-point multiply-add long instruction widens the odd-numbered 16-bit BFLOAT16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

Unlike the BFLOAT16 matrix multiplication and dot product instructions, this instruction performs a fused multiply-add that honors all of the FPCR bits that apply to single-precision arithmetic. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 0 0 1 0 0 0 1 1 1 | Zm | 1 0 0 0 0 1 | Zn | Zda |
| op2 |


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + 1, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, 2 * e + 1, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFMLALT (indexed)

BFLOAT16 floating-point multiply-add long to single-precision (top, indexed).

This BFLOAT16 floating-point multiply-add long instruction widens the odd-numbered 16-bit BFLOAT16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

Unlike the BFLOAT16 matrix multiplication and dot product instructions, this instruction performs a fused multiply-add that honors all of the FPCR bits that apply to single-precision arithmetic. It can also generate a floating-point exception that causes cumulative exception bits in the FPSR to be set, or a synchronous exception to be taken, depending on the enable bits in the FPCR.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

```
0 0 1 1 1 0 1 0 0 1 1 1 | i3h | Zm 0 1 0 0 | i3l 1 | Zn | Zda
```


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
<imm> Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 32;
integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(32) element1 = Elem[operand1, 2 * e + 1, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, s, 16] : Zeros(16);
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BFMLA

BFLOAT16 floating-point matrix multiply-accumulate.

This BFLOAT16 floating-point (BF16) matrix multiply-accumulate instruction multiplies the 2×4 matrix of BF16 values held in each 128-bit segment of the first source vector by the 4×2 BF16 matrix in the corresponding segment of the second source vector. The resulting 2×2 single-precision (FP32) matrix product is then destructively added to the FP32 matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 4-way dot product per destination element.

This instruction is unpredicated and vector length agnostic.

All floating-point calculations performed by this instruction are performed with the following behaviors, irrespective of the value in FPCR:

* Uses the non-IEEE 754 Round-to-Odd mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
* The cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC and IOC) are not modified.
* Trapped floating-point exceptions are disabled, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE and IOE) are all zero.
* Only the Default NaN is generated, as if FPCR.DN == 1.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.


if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;

integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();

integer segments = VL DIV 128;

bits(VL) operand1 = Z[n];

bits(VL) operand2 = Z[m];

bits(VL) operand3 = Z[da];

bits(VL) result;

bits(128) op1, op2;

bits(128) res, addend;

for s = 0 to segments-1

    op1 = Elem[operand1, s, 128];

    op2 = Elem[operand2, s, 128];

    addend = Elem[operand3, s, 128];

    res = BFMatMulAdd(addend, op1, op2);

    Elem[result, s, 128] = res;

Z[da] = result;

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE.
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BGRP

Group bits to right or left as selected by bitmask.

This instruction separates bits in each element of the first source vector by gathering from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, and from positions indicated by zero bits to the highest-numbered bits of the destination element, preserving the bit order within each group. This instruction is unpredicated. ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  |  size | 0  | Zm | 1  | 0  | 1  | 1  | 0  | Zn |  |  | Zd |<Zd><T>, <Zn><T>, <Zm><T>

if !HaveSVE2BitPerm() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) data = Z[n];
bits(VL) mask = Z[m];
bits(VL) result;
for e = 0 to elements - 1
    Elem[result, e, esize] = BitGroup(Elem[data, e, esize], Elem[mask, e, esize]);
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BIC (immediate)

Bitwise clear bits using immediate (unpredicated).

Bitwise clear bits using immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of \texttt{AND (immediate)}. This means:

- The encodings in this description are named to match the encodings of \texttt{AND (immediate)}.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of \texttt{AND (immediate)} gives the operational pseudocode for this instruction.

\[
\begin{array}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & \text{imm13} & \text{Zdn} \\
\end{array}
\]

\texttt{BIC <Zdn>.<T>, <Zdn>.<T>, #<const>}

is equivalent to

\texttt{AND <Zdn>.<T>, <Zdn>.<T>, #(<const> - 1)}

\textbf{Assembler Symbols}

- \texttt{<Zdn>} is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
- \texttt{<T>} is the size specifier, encoded in “\text{imm13<12>:imm13<5:0>}”:

<table>
<thead>
<tr>
<th>\text{imm13&lt;12&gt;}</th>
<th>\text{imm13&lt;5:0&gt;}</th>
<th>\text{&lt;T&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

- \texttt{<const>} is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “\text{imm13}” field.

\textbf{Operation}

The description of \texttt{AND (immediate)} gives the operational pseudocode for this instruction.

\textbf{Operational information}

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is \textbf{UNPREDICTABLE}:

- The \texttt{MOVPRFX} instruction must be unpredicated.
- The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BIC, BICS (predicates)

Bitwise clear predicates.

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | Pm | 0  | 1  | Pg | 0  | Pn | 1  | Pd |


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | Pm | 0  | 1  | Pg | 0  | Pn | 1  | Pd |


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
    P[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
BIC (vectors, predicated)

Bitwise clear vectors (predicated).

Bitwise AND inverted active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Pg</th>
<th></th>
<th>Zm</th>
<th></th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

BIC <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 AND (NOT element2);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**BIC (vectors, unpredicated)**

Bitwise clear vectors (unpredicated).

Bitwise AND inverted all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Zn</td>
<td>0</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BIC $<Zd>_D$, $<Zn>_D$, $<Zm>_D$

if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

$<Zd>$ Is the name of the destination scalable vector register, encoded in the “Zd” field.

$<Zn>$ Is the name of the first source scalable vector register, encoded in the “Zn” field.

$<Zm>$ Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

CheckSVEEnabled();
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
Z[d] = operand1 AND (NOT operand2);

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
BRKA, BRKAS

Break after first true condition.

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Pg | 0 | Pn | M | Pd |

BRKA <Pd>.B, <Pg>/<ZM>, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d =UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;

Setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | Pg | 0 | Pn | 0 | Pd |

BRKAS <Pd>.B, <Pg>/Z, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<ZM> Is the name of the source scalable predicate register, encoded in the "Pn" field.

<table>
<thead>
<tr>
<th>0</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = if !break then '1' else '0';
        break = break || element;
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';
    
if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
    P[d] = result;
```

BRKB, BRKBS

Break before first true condition.

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Pg | 0 | Pn | 0 | M | Pd |

BRKB <Pd>.B, <Pg>/<ZM>, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;

Setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | Pg | 0 | Pn | 0 | 0 | Pd |

BRKBS <Pd>.B, <Pg>/Z, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<ZM> Is the predication qualifier, encoded in “M”:

<table>
<thead>
<tr>
<th>M</th>
<th>&lt;ZM&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[n];
bits(PL) operand2 = P[d];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
  boolean element = ElemP[operand, e, esize] == '1';
  if ElemP[mask, e, esize] == '1' then
    break = break || element;
  elsif merging then
    ElemP[result, e, esize] = ElemP[operand2, e, esize];
  else
    ElemP[result, e, esize] = '0';
  end

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
  P[d] = result;
BRKN, BRKNS

Propagate break to next partition.

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0   0   1   0   1   0   1   0   0   1   1   0   0   0   0   1   | Pg | 0 |
| 0   0   1   0   1   0   1   0   0   1   1   0   0   0   0   1   | Pn | 0 |
|                                          | Pdm |


if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = FALSE;

Setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 0   0   1   0   1   0   1   0   1   0   1   0   1   0   0   0   0   1   | Pg | 0 |
| 0   0   1   0   1   0   1   0   0   1   1   0   0   0   0   1   | Pn | 0 |
|                                          | Pdm |


if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = TRUE;

Assembler Symbols

<Pdm> Is the name of the second source and destination scalable predicate register, encoded in the “Pdm” field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[dm];
bits(PL) result;
if LastActive(mask, operand1, 8) == '1' then
  result = operand2;
else
  result = Zeros();
if setflags then
  PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm] = result;
BRKPA, BRKPAS

Break after first true condition, propagating from previous partition.

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | 0 0 0 0 | Pm | 1 1 | Pg | 0 | Pn | 0 | Pd
```


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | 0 1 0 0 | Pm | 1 1 | Pg | 0 | Pn | 0 | Pd
```


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
  if ElemP[mask, e, 8] == '1' then
    ElemP[result, e, 8] = if last then '1' else '0';
    last = last && (ElemP[operand2, e, 8] == '0');
  else
    ElemP[result, e, 8] = '0';

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
BRKPB, BRKPBS

Break before first true condition, propagating from previous partition.

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BRKPB \(<Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>S</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BRKPBS \(<Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

\(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.
\(<Pg>\) Is the name of the governing scalable predicate register, encoded in the "Pg" field.
\(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.
\(<Pm>\) Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

`CheckSVEEnabled();`
integer elements = `VL` DIV `esize`;
bits(`PL`) mask = `P`[g];
bits(`PL`) operand1 = `P`[n];
bits(`PL`) operand2 = `P`[m];
bits(`PL`) result;
boolean last = (`LastActive`(mask, operand1, 8) == '1');

for e = 0 to elements-1
  if `ElemP`[mask, e, 8] == '1' then
    last = last && (`ElemP`[operand2, e, 8] == '0');
  else
    `ElemP`[result, e, 8] = '0';
if setflags then
  `PSTATE.<N,Z,C,V>` = `PredTest`(mask, result, `esize`);
`P`[d] = result;
BSL1N

Bitwise select with first input inverted.

Selects bits from the inverted first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 0 1 1</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

CheckSVEEnabled();
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[k];

Z[dn] = (NOT(operand1) AND operand3) OR (operand2 AND NOT(operand3));

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
BSL2N

Bitwise select with second input inverted.

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the inverted second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

CheckSVEEnabled();
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[k];
Z[dn] = (operand1 AND operand3) OR (NOT(operand2) AND NOT(operand3));

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Bitwise select.

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.

```plaintext
```

```plaintext
if !HaveSVE2() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

### Assembler Symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zk>` is the name of the third source scalable vector register, encoded in the "Zk" field.

### Operation

```plaintext
CheckSVEEnabled();
```

```plaintext
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[k];
```

```plaintext
Z[dn] = (operand1 AND operand3) OR (operand2 AND NOT(operand3));
```

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CADD

Complex integer add with rotate.

Add the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±j beforehand. Destructively place the results in the corresponding elements of the first source vector. This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 0 1 0 0 0 1 0 1 |  size  0 0 0 0 0 0 | 1 1 0 1 1 |  rot  Zm | Zdn
```

CADD <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

**Assembler Symbols**

- `<Zdn>`: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<const>`: Is the const specifier, encoded in "rot”:

```
<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#90</td>
</tr>
<tr>
<td>1</td>
<td>#270</td>
</tr>
</tbody>
</table>
```
Operation

\begin{verbatim}
CheckSVEEnabled();
integer pairs = VL \div (2 * esize);
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for p = 0 to pairs-1
    integer acc_r  = SInt(Elem[operand1, 2 * p + 0, esize]);
    integer acc_i  = SInt(Elem[operand1, 2 * p + 1, esize]);
    integer elt2_r = SInt(Elem[operand2, 2 * p + 0, esize]);
    integer elt2_i = SInt(Elem[operand2, 2 * p + 1, esize]);
    if sub_i then
        acc_r = acc_r - elt2_i;
        acc_i = acc_i + elt2_r;
    if sub_r then
        acc_r = acc_r + elt2_i;
        acc_i = acc_i - elt2_r;
    Elem[result, 2 * p + 0, esize] = acc_r<esize-1:0>;
    Elem[result, 2 * p + 1, esize] = acc_i<esize-1:0>;

Z[dn] = result;
\end{verbatim}

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Complex integer dot product.

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element which overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

* If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
* If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
* If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
* If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

### Assembler Symbols

- **<Zda>** Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- **<T>** Is the size specifier, encoded in “size<0>”:
  
<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the first source scalable vector register, encoded in the “Zn” field.
- **<Tb>** Is the size specifier, encoded in “size<0>”:
  
<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
</tbody>
</table>

- **<Zm>** Is the name of the second source scalable vector register, encoded in the “Zm” field.
- **<const>** Is the const specifier, encoded in “rot”:

```plaintext
CDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>, <const>
```

```plaintext
if !HaveSVE2() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);
```
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer elt1_r = SInt(Elem[operand1, 4 * e + 2 * i + 0, esize DIV 4]);
        integer elt1_i = SInt(Elem[operand1, 4 * e + 2 * i + 1, esize DIV 4]);
        integer elt2_a = SInt(Elem[operand2, 4 * e + 2 * i + sel_a, esize DIV 4]);
        integer elt2_b = SInt(Elem[operand2, 4 * e + 2 * i + sel_b, esize DIV 4]);
        if sub_i then
            res = res + (elt1_r * elt2_a) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
    Elem[result, e, esize] = res;
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CDOT (indexed)

Complex integer dot product (indexed).

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element which overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

* If the rotation is 0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
* If the rotation is 90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
* If the rotation is 180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
* If the rotation is 270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

The indexed form of these instructions select a single pair of complex numbers within each 128-bit segment of the second source vector as the multiplier for all pairs of complex numbers within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index which selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment, encoded in 1 or 2 bits depending on the size of the complex number pair.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: 32-bit and 64-bit.

### 32-bit

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|----------------|
| 0   1   0   0   1   0   0   1 | 0   1   1   i2 | Zm   0   1   0   0 | rot | Zn   Zda |
```


```java
if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);
```

### 64-bit

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|----------------|
| 0   1   0   0   1   0   0   1 | 1   1   i1 | Zm   0   1   0   0 | rot | Zn   Zda |
```

CDOT (indexed)
if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
<imm> For the 32-bit variant: is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.
For the 64-bit variant: is the immediate index of a quadtuplet of four 16-bit elements within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.
<const> Is the const specifier, encoded in “rot”:

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 1
    integer elt1_r = SInt(Elem[operand1, 4 * e + 2 * i + 0, esize DIV 4]);
    integer elt1_i = SInt(Elem[operand1, 4 * e + 2 * i + 1, esize DIV 4]);
    integer elt2_a = SInt(Elem[operand2, 4 * s + 2 * i + sel_a, esize DIV 4]);
    integer elt2_b = SInt(Elem[operand2, 4 * s + 2 * i + sel_b, esize DIV 4]);
    if sub_i then
      res = res + (elt1_r * elt2_a) - (elt1_i * elt2_b);
    else
      res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
    Elem[result, e, esize] = res;
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**CLASTA (scalar)**

Conditionally extract element after last to general-purpose register.

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 | size | 1 1 0 0 0 | 0 1 0 1 | Pg | Zm | Rdn
```

CLASTA \(<R><dn>, <Pg>, <R><dn>, <Zm><T>\)

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
integer csize = if esize < 64 then 32 else 64;
boolean isBefore = FALSE;

Assembler Symbols

\(<R>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

\(<dn>\) Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the source scalable vector register, encoded in the "Zm" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

**Operation**

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = X[dn];
bits(VL) operand2 = Z[m];
bits(csize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = ZeroExtend(operand1);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
        result = ZeroExtend(Elem(operand2, last, esize));

X[dn] = result;
```

CLASTA (scalar)
CLASTA (SIMD&FP scalar)

Conditionally extract element after last to SIMD&FP scalar register.

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------------------|---------|---------|---------|
| 0 0 0 0 0 1 0 1 | size | 1 0 1 0 1 | 0 1 | 0 0 | Pg | Zm | Vdn |

CLASTA \(<V><dn>, <Pg>, <V><dn>, <Zm>..<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<dn>\) Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the source scalable vector register, encoded in the "Zm" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = V[dn];
bits(VL) operand2 = Z[m];
bits(esize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = ZeroExtend(operand1);
else
    if !isBefore then
        last = last + 1;
    if last >= elements then last = 0;
    result = Elem(operand2, last, esize);

V[dn] = result;
CLASTA (vectors)

Conditionally extract element after last to vector register.

From the second source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  |

CLASTA \(<\text{Zdn}>.\langle T\rangle, \langle Pg\rangle, \langle Zdn\rangle.\langle T\rangle, \langle Zm\rangle.\langle T\rangle\)

```java
if !\text{HaveSVE}() then UNDEFINED;
integer esize = 8 \ll \text{UInt}(\text{size});
integer g = \text{UInt}(\text{Pg});
integer dn = \text{UInt}(\text{Zdn});
integer m = \text{UInt}(\text{Zm});
boolean isBefore = FALSE;
```

Assembler Symbols

- \(<\text{Zdn}>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- \(<\text{T}>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;\text{T}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```java
\text{CheckSVEEnabled}();
integer elements = \text{VL} \div \text{esize};
\text{bits(PL)} mask = \text{P}[g];
\text{bits(VL)} operand1 = \text{Z}[dn];
\text{bits(VL)} operand2 = \text{Z}[m];
\text{bits(VL)} result;
integer last = \text{LastActiveElement}(mask, esize);
if last < 0 then
  result = operand1;
else
  if !isBefore then
    last = last + 1;
  if last \geq elements then last = 0;
  for e = 0 to elements-1
    \text{Elem}[result, e, esize] = \text{Elem}[operand2, last, esize];
\text{Z}[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**CLASTB (scalar)**

Conditionally extract last element to general-purpose register.

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.

```plaintext
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 1 | size | 1 1 0 0 0 1 | Pg | Zm | Rdn |
```

CLASTB `<R><dn>`, `<Pg>`, `<R><dn>`, `<Zm>`. `<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
integer csize = if esize < 64 then 32 else 64;
boolean isBefore = TRUE;

**Assembler Symbols**

- `<R>` Is a width specifier, encoded in “size”:
  - **size** | `<R>`
    - 01 | W
    - x0 | W
    - 11 | X

- `<dn>` Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Zm>` Is the name of the source scalable vector register, encoded in the "Zm" field.

- `<T>` Is the size specifier, encoded in “size”:
  - **size** | `<T>`
    - 00 | B
    - 01 | H
    - 10 | S
    - 11 | D

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = X[dn];
bits(VL) operand2 = Z[m];
bits(csize) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
  result = ZeroExtend(operand1);
else
  if !isBefore then
    last = last + 1;
  if last >= elements then last = 0;
  result = ZeroExtend(Elem[operand2, last, esize]);
X[dn] = result;
```

CLASTB (scalar)
CLASTB (SIMD&FP scalar)

Conditionally extract last element to SIMD&FP scalar register.

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 1 1 0 1 1 1 0 0 | Pg | Zm | Vdn |

CLASTB \(<V><dn>, <Pg>, <V><dn>, <Zm>.<T>\)

if !HaveSVE() then UNDEFINED;
integer esize = 8 \(<\text{UInt}(size)\);
integer g = \text{UInt}(Pg);
integer dn = \text{UInt}(Vdn);
integer m = \text{UInt}(Zm);
boolean isBefore = TRUE;

Assembler Symbols

- \(<V>\) Is a width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<dn>\) Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.
- \(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<Zm>\) Is the name of the source scalable vector register, encoded in the "Zm" field.
- \(<T>\) Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

\(\text{CheckSVEEnabled}();\)
integer elements = VL \(<\text{DIV} \text{esize};\)
bits(PL) mask = \text{P}[g];
bits(esize) operand1 = \text{V}[dn];
bits(VL) operand2 = \text{Z}[m];
bits(esize) result;
integer last = \text{LastActiveElement}(mask, esize);
if last < 0 then
  result = \text{ZeroExtend}(operand1);
else
  if !isBefore then
    last = last + 1;
    if last >= elements then last = 0;
    result = \text{Elem}(operand2, last, esize);
  \text{V}[dn] = result;
**CLASTB (vectors)**

Conditionally extract last element to vector register.

From the second source vector register extract the last active element, and then replicate that element to destructively fill the destination and first source vector.

If there are no active elements then leave the destination and source vector unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 0 0 1 1 0 0 Pg Zm Zdn</td>
</tr>
</tbody>
</table>

**CLASTB** `<Zdn>.<T>`, `<Pg>`, `<Zdn>.<T>`, `<Zm>.<T>`

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;
```

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer last = LastActiveElement(mask, esize);
if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
    if last >= elements then last = 0;
    for e = 0 to elements-1
        Elem[result, e, esize] = Elem[operand2, last, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CLS

Count leading sign bits (predicated).

Count leading sign bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 1 0 0 | size | 0 1 1 | 0 0 | 1 0 1 | Pg | Zn | Zd |

CLS <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = CountLeadingSignBits(element)<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
     ○ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
     ○ The values of the NZCV flags.
   • The response of this instruction to asynchronous exceptions does not vary based on:
     ○ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
     ○ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
   • The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Count leading zero bits (predicated).

Count leading zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |

CLZ <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = CountLeadingZeroBits(element)<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMLA (vectors)

Complex integer multiply-add with rotate.

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the results in the corresponding elements of the addend vector. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
CMLA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>
```

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for p = 0 to pairs-1
   integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
   integer elt2_a = SInt(Elem[operand2, 2 * p + sel_a, esize]);
   integer elt2_b = SInt(Elem[operand2, 2 * p + sel_b, esize]);
   bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
   bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
   integer product_r = elt1_a * elt2_a;
   integer product_i = elt1_a * elt2_b;
   if sub_r then
       Elem[result, 2 * p + 0, esize] = elt3_r - product_r;
   else
       Elem[result, 2 * p + 0, esize] = elt3_r + product_r;
   if sub_i then
       Elem[result, 2 * p + 1, esize] = elt3_i - product_i;
   else
       Elem[result, 2 * p + 1, esize] = elt3_i + product_i;
Z[da] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
     ◦ The values of the data supplied in any of its registers.
     ◦ The values of the NZCV flags.
   • The response of this instruction to asynchronous exceptions does not vary based on:
     ◦ The values of the data supplied in any of its registers.
     ◦ The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
   • The MOVPRFX instruction must be unpredicated.
   • The MOVPRFX instruction must specify the same destination register as this instruction.
   • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CMLA (indexed)

Complex integer multiply-add with rotate (indexed).

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation. Then add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the results in the corresponding elements of the addend vector. This instruction is unpredicated. These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: 16-bit and 32-bit

16-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 1 0 i2</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 1</td>
</tr>
</tbody>
</table>

CMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Is the const specifier, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>

Operation

```
CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for p = 0 to pairs-1
    integer segmentbase = p - (p MOD pairspersegment);
    integer s = segmentbase + index;
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r = elt1_a * elt2_a;
    integer product_i = elt1_a * elt2_b;
    if sub_r then
        Elem[result, 2 * p + 0, esize] = elt3_r - product_r;
    else
        Elem[result, 2 * p + 0, esize] = elt3_r + product_r;
    if sub_i then
        Elem[result, 2 * p + 1, esize] = elt3_i - product_i;
    else
        Elem[result, 2 * p + 1, esize] = elt3_i + product_i;
Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CMP<cc> (immediate)

Compare vector to immediate.

Compare active integer elements in the source vector with an immediate, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, HI, HS, LE, LO, LS, LT or NE.

It has encodings from 10 classes: Equal, Greater than, Greater than or equal, Higher, Higher or same, Less than, Less than or equal, Lower, Lower or same and Not equal.

Equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | size | 0 | imm5 | 1 | 0 | 0 | Pg | Zn | 0 | Pd |

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Greater than

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | size | 0 | imm5 | 1 | 0 | 0 | Pg | Zn | 1 | Pd |

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Greater than or equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | size | 0 | imm5 | 0 | 0 | 0 | Pg | Zn | 0 | Pd |

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
Higher

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>imm7</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0</td>
<td>0</td>
<td>Pg</td>
<td>Zn</td>
<td>1</td>
<td>Pd</td>
</tr>
</tbody>
</table>

CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEcmp op = Cmp_GT;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Higher or same

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>imm7</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0</td>
<td>0</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>Pd</td>
</tr>
</tbody>
</table>

CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEcmp op = Cmp_GE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Less than

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>imm5</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0</td>
<td>0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>Pd</td>
</tr>
</tbody>
</table>

CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEcmp op = Cmp_LT;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Less than or equal

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>imm5</th>
<th>Pg</th>
<th>Zn</th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0</td>
<td>0 0 1</td>
<td>Pg</td>
<td>Zn</td>
<td>1</td>
<td>Pd</td>
</tr>
</tbody>
</table>
CMPLE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Lower

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | size | 1  | imm7 | 1  | Pg | Zn | 0  | 0  | Pd |

CMPL0 <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Lower or same

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | size | 1  | imm7 | 1  | Pg | Zn | 1  | Pd |

CMPLS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;

Not equal

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | size | 0  | imm5 | 1  | 0  | 0  | Pg | Zn | 1  | Pd |

CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;

Assembler Symbols

<Pd> is the name of the destination scalable predicate register, encoded in the "Pd" field.
Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Is the name of the source scalable vector register, encoded in the "Zn" field.

For the equal, greater than, greater than or equal, less than, less than or equal and not equal variant: is the signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

For the higher, higher or same, lower and lower or same variant: is the unsigned immediate operand, in the range 0 to 127, encoded in the “imm7” field.

### Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits<PL> mask = P[g];
bits<VL> operand1 = Z[n];
bits<PL> result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        boolean cond;
        case op of
            when Cmp_EQ  cond = element1 == imm;
            when Cmp_NE  cond = element1 != imm;
            when Cmp_GE  cond = element1 >= imm;
            when Cmp_LT  cond = element1 <  imm;
            when Cmp_GT  cond = element1 >  imm;
            when Cmp_LE  cond = element1 <= imm;
        ElemP[result, e, esize] = if cond then '1' else '0';
        else
            ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CMP<cc> (wide elements)

Compare vector to 64-bit wide elements.

Compare active integer elements in the first source vector with overlapping 64-bit doubleword elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, HI, HS, LE, LO, LS, LT or NE.

It has encodings from 10 classes: Equal, Greater than, Greater than or equal, Higher, Higher or same, Less than, Less than or equal, Lower, Lower or same and Not equal.

Equal

```
0 0 1 0 0 1 0 0 | size 0 | Zm 0 0 1 | Pg 0 1 0 | Zn 0 1 | Pd
```

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>..<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
boolean unsigned = FALSE;

Greater than

```
0 0 1 0 0 1 0 0 | size 0 | Zm 0 1 0 | Pg 0 0 1 | Zn 1 0 | Pd
```

CMPIG <Pd>.<T>, <Pg>/Z, <Zn>..<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = FALSE;

Greater than or equal

```
0 0 1 0 0 1 0 0 | size 0 | Zm 0 1 0 | Pg 0 1 0 | Zn 0 0 | Pd
```


if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = FALSE;

Higher

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 0 | size | 0 | Zm | 1 | 1 | 0 | Pg | Zn | 1 | Pd |
| | U | lt | ne |

CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = TRUE;

Higher or same

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 0 | size | 0 | Zm | 1 | 1 | 0 | Pg | Zn | 0 | Pd |
| | U | lt | ne |

CMPSH <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = TRUE;

Less than

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 0 | size | 0 | Zm | 0 | 1 | 1 | Pg | Zn | 0 | Pd |
| | U | lt | ne |

CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = FALSE;
Less than or equal

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 1 & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 1 & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

CMPLE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = FALSE;

Lower

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 1 & 1 & 1 & \text{Pg} & \text{Zn} & 0 & \text{Pd} \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 1 & 1 & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

CMPL0 <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = TRUE;

Lower or same

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 1 & 1 & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 1 & 0 & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

CMPLS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = TRUE;

Not equal

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 0 & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & \text{Zm} & 0 & 0 & 1 & \text{Pg} & \text{Zn} & 1 & \text{Pd} \\
\end{array}
\]
if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
boolean unsigned = FALSE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(PL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, (e * esize) DIV 64, 64], unsigned);
    if ElemP[mask, e, esize] == '1' then
        boolean cond;
        case op of
        when Cmp_EQ cond = element1 == element2;
        when Cmp_NE cond = element1 != element2;
        when Cmp_GE cond = element1 >= element2;
        when Cmp_LT cond = element1 <  element2;
        when Cmp_GT cond = element1 >  element2;
        when Cmp_LE cond = element1 <= element2;
        ElemP[result, e, esize] = if cond then '1' else '0';
        else
            ElemP[result, e, esize] = '0';
    end case;
end for;
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
CMP<cc> (vectors)

Compare vectors.

Compare active integer elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, HI, HS or NE.

This instruction is used by the pseudo-instructions CMPLE (vectors), CMPLO (vectors), CMPLS (vectors), and CMPLT (vectors).

It has encodings from 6 classes: Equal, Greater than, Greater than or equal, Higher, Higher or same and Not equal.

**Equal**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0</td>
</tr>
</tbody>
</table>

CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
boolean unsigned = FALSE;

**Greater than**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0</td>
</tr>
</tbody>
</table>

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = FALSE;

**Greater than or equal**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 0</td>
</tr>
</tbody>
</table>
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = FALSE;

Higher

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = TRUE;

Higher or same

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

CMPI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = TRUE;

Not equal

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

CMPI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
Is the size specifier, encoded in “size”:

| size | 
|------|-
| 00   | B  
| 01   | H  
| 10   | S  
| 11   | D  

Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

Is the name of the first source scalable vector register, encoded in the “Zn” field.

Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```cpp
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(PL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if ElemP[mask, e, esize] == '1' then
    boolean cond;
    case op of
      when Cmp_EQ cond = element1 == element2;
      when Cmp_NE cond = element1 != element2;
      when Cmp_GE cond = element1 >= element2;
      when Cmp_LT cond = element1 <  element2;
      when Cmp_GT cond = element1 >  element2;
      when Cmp_LE cond = element1 <= element2;
        ElemP[result, e, esize] = if cond then '1' else '0';
    else
      ElemP[result, e, esize] = '0';
  end;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
CMPLE (vectors)

Compare signed less than or equal to vector, setting the condition flags.

Compare active signed integer elements in the first source vector being less than or equal to corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th>Zm</th>
<th></th>
<th></th>
<th>Pg</th>
<th>Zn</th>
<th></th>
<th>Pd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>size</td>
<td>0</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>Pd</td>
</tr>
</tbody>
</table>

CMPLE <Pd>..<T>, <Pg>/Z, <Zm>..<T>, <Zn>..<T>

is equivalent to

CMPGE <Pd>..<T>, <Pg>/Z, <Zn>..<T>, <Zm>..<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
CMPLO (vectors)

Compare unsigned lower than vector, setting the condition flags.

Compare active unsigned integer elements in the first source vector being lower than corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

• The encodings in this description are named to match the encodings of CMP<cc> (vectors).
• The assembler syntax is used only for assembly, and is not used on disassembly.
• The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

This instruction is equivalent to:

CMPHI <Pd>..<T>, <Pg>/Z, <Zm>..<T>, <Zn>..<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ○ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ○ The values of the NZCV flags.

  The response of this instruction to asynchronous exceptions does not vary based on:
  ○ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ○ The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**CMPLS (vectors)**

Compare unsigned lower or same as vector, setting the condition flags.

Compare active unsigned integer elements in the first source vector being lower than or same as corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the first (N), none (Z), last (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of **CMP<cc> (vectors)**. This means:

- The encodings in this description are named to match the encodings of **CMP<cc> (vectors)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **CMP<cc> (vectors)** gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**CMPLS <Pd>..<T>, <Pg>/Z, <Zm>..<T>, <Zn>..<T>**

is equivalent to

**CMPSH <Pd>..<T>, <Pg>/Z, <Zn>..<T>, <Zm>..<T>**

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of **CMP<cc> (vectors)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
CMPLT (vectors)

Compare signed less than vector, setting the condition flags.

Compare active signed integer elements in the first source vector being less than corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

CMPLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

Assembler Symbols

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CNOT

Logically invert boolean condition in vector (predicated).

Logically invert the boolean value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Boolean TRUE is any non-zero value in a source, and one in a result element. Boolean FALSE is always zero.

\[
\begin{array}{cccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & P & g & Z & n & Z & d
\end{array}
\]

CNOT <Zd>,<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = ZeroExtend(IsZeroBit(element), esize);

Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**CNT**

Count non-zero bits (predicated).

Count non-zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>size</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

CNT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

**Assembler Symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>`: Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>`: Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = BitCount(element<esize-1:0>);
Z[d] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CNTB, CNTD, CNTH, CNTW

Set scalar to multiple of predicate constraint element count.

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then places the result in the scalar destination.
The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).
Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword and Word

**Byte**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 0 0 0 1 0 0    | 0 0 1 0 imm4     | 1 1 1 0 0 0      |

size<1>size<0> Rd
```

CNTB <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Doubleword**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 0 0 0 1 0 1 0  | imm4 1 1 1 0 0 0 |

size<1>size<0> Rd
```

CNTD <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Halfword**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 0 0 0 1 0 1 0   | imm4 1 1 1 0 0 0 |

size<1>size<0> Rd
```

CNTH <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
CNTW <Xd>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
X[d] = (count * imm)<63:0>;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Set scalar to count of true predicate elements.

Counts the number of active and true elements in the source predicate and places the scalar result in the destination general-purpose register. Inactive predicate elements are not counted.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

CNTP <Xd>, <Pg>, <Pn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Rd);

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bins(PL) mask = P[g];
bins(PL) operand = P[n];
bits(64) sum = Zeros();

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' && ElemP[operand, e, esize] == '1' then
    sum = sum + 1;
X[d] = sum;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
Shuffle active elements of vector to the right and fill with zero.

Read the active elements from the source vector and pack them into the lowest-numbered elements of the destination vector. Then set any remaining elements of the destination vector to zero.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |

COMPACT <Zd>, <T>, <Pg>, <Zn>, <T>

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) result;
integer x = 0;
for e = 0 to elements-1
    Elem[result, e, esize] = Zeros();
    if Elem[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand1, e, esize];
        Elem[result, x, esize] = element;
        x = x + 1;
Z[d] = result;
**CPY (immediate, zeroing)**

Copy signed integer immediate to vector elements (zeroing).

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However, an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias **MOV (immediate, predicated, zeroing)**.

```
CPY <Zd>, <T>, #<imm>{, <shift>}
```

If `!HaveSVE()` then UNDEFINED;
If `size:sh == '001'` then UNDEFINED;
Integer `esize = 8 << UInt(size)`;
Integer `g = UInt(Pg)`;
Integer `d = UInt(Zd)`;
Boolean `merging = FALSE`;
Integer `imm = SInt(imm8)`;
If `sh == '1'` then `imm = imm << 8`;

**Assembler Symbols**

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<imm>` is a signed immediate in the range -128 to 127, encoded in the "imm8" field.
- `<shift>` is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":
  
<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) dest = Z[d];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = imm<esize-1:0>;
    elsif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros();
    
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
**CPY (immediate, merging)**

Copy signed integer immediate to vector elements (merging).

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the aliases **FMOV (zero, predicated)**, and **MOV (immediate, predicated, merging)**.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Pg</th>
<th>sh</th>
<th>imm8</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(PL) mask = \texttt{P}[g];
bits(VL) dest = \texttt{Z}[d];
bits(VL) result;

for e = 0 to elements-1
  if \texttt{ElemP}[mask, e, esize] == '1' then
    \texttt{Elem}[result, e, esize] = \texttt{imm}<\texttt{esize}-1:0>;
  elsif merging then
    \texttt{Elem}[result, e, esize] = \texttt{Elem}[dest, e, esize];
  else
    \texttt{Elem}[result, e, esize] = \texttt{Zeros}();

\texttt{Z}[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CPY (scalar)

Copy general-purpose register to vector elements (predicated).

Copy the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias MOV (scalar, predicated).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| CPY <Zd>, <T>, <Pg>/M, <R><n|SP> |

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Rn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) operand1;
if n == 31 then
   operand1 = SP[];
else
   operand1 = X[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
   if Elem[mask, e, esize] == '1'
      Elem[result, e, esize] = operand1<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
   • The execution time of this instruction is independent of:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CPY (SIMD&FP scalar)

Copy SIMD&FP scalar register to vector elements (predicated).

Copy the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This instruction is used by the alias MOV (SIMD&FP scalar, predicated).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| size                         | Pg            | Vn | Zd |
| 00 00 00 00 01 01            | 1 0 0 0 0 0   | 1 0 0 |

CPY <Zd>,<T>, <Pg>/M, <V><n>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Vn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the “Vn” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = V[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = operand1;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
◆ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
◆ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
CTERMEQ, CTERMNE

Compare and terminate loop.

Detect termination conditions in serialized vector loops. Tests whether the comparison between the scalar source operands holds true and if not tests the state of the !LAST condition flag (C) which indicates whether the previous flag-setting predicate instruction selected the last element of the vector partition.

The Z and C condition flags are preserved by this instruction. The N and V condition flags are set as a pair to generate one of the following conditions for a subsequent conditional instruction:
- GE (N=0 & V=0): continue loop (compare failed and last element not selected);
- LT (N=0 & V=1): terminate loop (last element selected);
- LT (N=1 & V=0): terminate loop (compare succeeded);

The scalar source operands are 32-bit or 64-bit general-purpose registers of the same size.

It has encodings from 2 classes: Equal and Not equal

### Equal

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 1 sz 1 Rm 0 0 1 0 0 0 Rn 0 0 0 0 ne
```

CTERMEQ <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVECmp op = Cmp_EQ;

### Not equal

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 1 sz 1 Rm 0 0 1 0 0 0 Rn 1 0 0 0 0 ne
```

CTERMNE <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVECmp op = Cmp_NE;

### Assembler Symbols

- **<R>** Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

- **<n>** Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

- **<m>** Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
bits(esize) operand1 = X[n];
bits(esize) operand2 = X[m];
integer element1 = UInt(operand1);
integer element2 = UInt(operand2);
boolean term;

    case op of
        when Cmp_EQ term = element1 == element2;
        when Cmp_NE term = element1 != element2;
    if term then
        PSTATE.N = '1';
        PSTATE.V = '0';
    else
        PSTATE.N = '0';
        PSTATE.V = (NOT PSTATE.C);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
DECB, DECD, DECH, DECW (scalar)

Decrement scalar by multiple of predicate constraint element count.

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword and Word

**Byte**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 0 0 0 1 1 imm4 1 1 1 0 0 1 pattern Rdn
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

**Doubleword**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 1 1 imm4 1 1 1 0 0 1 pattern Rdn
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

**Halfword**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 1 1 imm4 1 1 1 0 0 1 pattern Rdn
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```


**Assembler Symbols**

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>00x01</td>
<td>#uimm5</td>
</tr>
<tr>
<td>00x10</td>
<td>#uimm5</td>
</tr>
<tr>
<td>00x00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.

**Operation**

```plaintext
DECW <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**DECD, DECH, DECW (vector)**

Decrement vector by multiple of predicate constraint element count.

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:
- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: [Doubleword](#), [Halfword](#) and [Word](#)

**Doubleword**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 0 0 1 0 0 0 1 1 1 imm4 1 1 0 0 0 1 pattern Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
<td>D</td>
</tr>
</tbody>
</table>

```assembly
DEC <Zdn>.D{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Halfword**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 0 0 1 0 0 0 1 1 1 imm4 1 1 0 0 0 1 pattern Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
<td>D</td>
</tr>
</tbody>
</table>

```assembly
DECH <Zdn>.H{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Word**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 0 0 1 0 0 1 0 1 1 imm4 1 1 0 0 0 1 pattern Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
<td>D</td>
</tr>
</tbody>
</table>

```assembly
DECW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Assembler Symbols**

<Zdn> is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
if (CheckSVEEnabled()) {
    integer elements = VL(DIV esize);
    integer count = DecodePredCount(pat, esize);
    bits(VL) operand1 = Z[dn];
    bits(VL) result;
    for (e = 0 to elements-1)
        Elem[result, e, esize] = Elem[operand1, e, esize] - (count * imm);
    Z[dn] = result;
}
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
DECP (scalar)

Decrement scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

DECP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Pm> Is the name of the source scalable predicate register, encoded in the “Pm” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) operand1 = X[dn];
bits.PLL operand2 = P[m];
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

X[dn] = operand1 - count;
**DECP (vector)**

Decrement vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

DECP <Zdn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;

for e = 0 to elements-1
  Elem[result, e, esize] = Elem[operand1, e, esize] - count;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
DUP (immediate)

Broadcast signed immediate to vector elements (unpredicated).

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is “#<simm8>, LSL #8”. However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as “#0, LSL #8”.

This instruction is used by the aliases FMOV (zero, unpredicated), and MOV (immediate, unpredicated).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | sh | imm8 | Zd |

DUP <Zd>.<T>, #<imm>{, <shift>}

if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Zd);
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(VL) result = Replicate(imm<esize-1:0>);
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
DUP (scalar)

Broadcast general-purpose register to vector elements (unpredicated).

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (scalar, unpredicated).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 1  | size | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | Rn | Zd |

DUP <Zd>.<T>, <R><n|SP>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) operand;
if n == 31 then
    operand = SP[];
else
    operand = X[n];
bits(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = operand<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
DUP (indexed)

Broadcast indexed element to vector (unpredicated).

Unconditionally broadcast the indexed source vector element into each element of the destination vector. This instruction is unpredicated.

The immediate element index is in the range of 0 to 63 (bytes), 31 (halfwords), 15 (words), 7 (doublewords) or 3 (quadwords). Selecting an element beyond the accessible vector length causes the destination vector to be set to zero.

This instruction is used by the alias MOV (SIMD&FP scalar, unpredicated).

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 1 |imm2| 1 |tsz| 0 0 1 0 0 0 |Zn| Zd

DUP <Zd>.<T>, <Zn>.<T>[<imm>]

if !HaveSVE() then UNDEFINED;

bits(7) imm = imm2:tsz;

case tsz of
  when '00000' UNDEFINED;
  when '10000' esize = 128; index = UInt(imm<6:5>);
  when 'xx100' esize = 64; index = UInt(imm<6:4>);
  when 'xxx10' esize = 32; index = UInt(imm<6:3>);
  when 'xxxx1' esize = 16; index = UInt(imm<6:2>);
  when 'xxxxx1' esize = 8; index = UInt(imm<6:1>);

integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tsz”:

<table>
<thead>
<tr>
<th>tsz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
<tr>
<td>10000</td>
<td>Q</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in “imm2:tsz”.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (SIMD&amp;FP scalar, unpredicated)</td>
<td>BitCount(imm2:tsz) == 1</td>
</tr>
<tr>
<td>MOV (SIMD&amp;FP scalar, unpredicated)</td>
<td>BitCount(imm2:tsz) &gt; 1</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
bits(esize) element;

if index >= elements then
    element = Zeros();
else
    element = Elem[operand1, index, esize];
result = Replicate(element);
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
DUPM

Broadcast logical bitmask immediate to vector (unpredicated).

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This instruction is used by the alias MOV (bitmask immediate).

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|--------------------------|
| 0 0 0 0 0 1 0 1 1 1 0 0 0 0 | imm13                   |

DUPM <Zd>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer d = UINT(Zd);
bits(esize) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (bitmask immediate)</td>
<td>SVEMoveMaskPreferred(imm13)</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(VL) result = Replicate(imm);
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Bitwise exclusive OR with inverted immediate (unpredicated).

Bitwise exclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of **EOR (immediate)**. This means:

- The encodings in this description are named to match the encodings of **EOR (immediate)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **EOR (immediate)** gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>Imm13&lt;12&gt;</th>
<th>Imm13&lt;5:0&gt;</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

**<Zdn>**

Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

**<T>**

Is the size specifier, encoded in “imm13<12>:imm13<5:0>“:

**<const>**

Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

**Operation**

The description of **EOR (immediate)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
EOR3

Bitwise exclusive OR of three vectors.

Bitwise exclusive OR the corresponding elements of all three source vectors, and destructively place the results in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & Zm & 0 & 0 & 1 & 1 & 1 & 0 & Zk & Zdn
\end{array}
\]


if !HaveSVE2() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
n integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk> Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

CheckSVEEnabled();
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[k];
Z[dn] = operand1 EOR operand2 EOR operand3;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

  • The MOVPRFX instruction must be unpredicated.
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
EOR, EORS (predicates)

Bitwise exclusive OR predicates.

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the aliases NOTS, and NOT (predicate).

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags.

Not setting the condition flags

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 0 0 0 1 0 1 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 1 0 0 1 0 1 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTS</td>
<td>Pm == Pg</td>
</tr>
<tr>
<td>NOT (predicate)</td>
<td>Pm == Pg</td>
</tr>
</tbody>
</table>
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 EOR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
    P[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
EOR (vectors, predicated)

Bitwise exclusive OR vectors (predicated).

Bitwise exclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
0 0 0 0 0 1 0 0 size 0 1 1 0 0 1 0 0 Pg Zm Zdn
```

EOR <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>,<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 EOR element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**EOR (immediate)**

Bitwise exclusive OR with immediate (unpredicated).

Bitwise exclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction **EON**.

<table>
<thead>
<tr>
<th>EOR &lt;Zdn&gt;.&lt;T&gt;, &lt;Zdn&gt;.&lt;T&gt;, #&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>if !HaveSVE() then UNDEFINED;</td>
</tr>
<tr>
<td>integer dn = UInt(Zdn);</td>
</tr>
<tr>
<td>bits(64) imm;</td>
</tr>
<tr>
<td>(imm, -) = DecodeBitMasks(imm13&lt;12&gt;, imm13&lt;5:0&gt;, imm13&lt;11:6&gt;, TRUE);</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

**Operation**

**CheckSVEEnabled();**

integer elements = VL DIV 64;

bits(VL) operand = Z[dn];

bits(VL) result;

for e = 0 to elements-1

  bits(64) element1 = Elem[operand, e, 64];

  Elem[result, e, 64] = element1 EOR imm;

Z[dn] = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
EOR (vectors, unpredicated)

Bitwise exclusive OR vectors (unpredicated).

Bitwise exclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 1   | 0   | 1   | 0   | 1   | Zm  | 0   | 0   | 1   | 1   | 0   | 0   | Zn  | Zd  |

EOR <Zd>.D, <Zn>.D, <Zm>.D

if !HaveSVE() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
Z[d] = operand1 EOR operand2;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
EORBT

Interleaving exclusive OR (bottom, top).

Interleaving exclusive OR between the even-numbered elements of the first source vector register and the odd-numbered elements of the second source vector register, placing the result in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged. This instruction is unpredicated.

```
0 1 0 0 1 0 1 size 0 Zm 1 0 0 1 0 0 Zn 1 Zd
```

EORBT <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[d];

for e = 0 to elements - 1
  bits(esize) element1 = Elem[operand1, 2*e + sel1, esize];
  bits(esize) element2 = Elem[operand2, 2*e + sel2, esize];
  Elem[result, 2*e + sel1, esize] = element1 EOR element2;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
EORTB

Interleaving exclusive OR (top, bottom).

Interleaving exclusive OR between the odd-numbered elements of the first source vector register and the even-numbered elements of the second source vector register, placing the result in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged. This instruction is unpredicated.

\[
\begin{array}{ccccccccccccccccc}
\hline
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & Zm & 1 & 0 & 0 & 1 & 0 & 1 & \text{Zn} & 0 & 1 & 0 & 1 \text{Zm} & \text{tb} & \text{Zd}
\end{array}
\]

EORTB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 0;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, 2*e + sel1, esize];
    bits(esize) element2 = Elem[operand2, 2*e + sel2, esize];
    Elem[result, 2*e + sel1, esize] = element1 EOR element2;

Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
EORV

Bitwise exclusive OR reduction to scalar.

Bitwise exclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------|-------------------------------|-------------------------------|-------------------------------|
| 0 0 0 0 0 0 1 0 0 | 0 1 1 0 0 1 0 1 0 | Pg | Zn | Vd |

EORV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) result = Zeros(esize);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == ‘1’ then
        result = result EOR Elem[operand, e, esize];
V[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
The values of the NZCV flags.
EXT

Extract vector from pair of vectors.

Copy the indexed byte up to the last byte of the first source vector to the bottom of the result vector, then fill the remainder of the result starting from the first byte of the second source vector. The result is placed destructively in the destination and first source vector, or constructively in the destination vector. This instruction is unpredicated.

An index that is greater than or equal to the vector length in bytes is treated as zero, resulting in the first source vector being copied to the result unchanged.

It has encodings from 2 classes: Constructive and Destructive

Constructive

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 0 1 1</td>
</tr>
</tbody>
</table>


if !HaveSVEZ() then UNDEFINED;
integer esize = 8;
integer dst = UInt(Zd);
integer s1 = UInt(Zn);
integer s2 = (s1 + 1) MOD 32;
integer position = UInt(imm8h:imm8l);

Destructive

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1 0 0 1</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dst = UInt(Zdn);
integer s1 = dst;
integer s2 = UInt(Zm);
integer position = UInt(imm8h:imm8l);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.
<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.
<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8h:imm8l" fields.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[s1];
bits(VL) operand2 = Z[s2];
bits(VL) result;

if position >= elements then
    position = 0;
position = position << 3;
bits(VL*2) concat = operand2 : operand1;
result = concat[position+VL-1:position>);
Z[dst] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point absolute difference (predicated).

Compute the absolute difference of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

FABD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

< Pg > Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
< Zm > Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
   bits(esize) element1 = Elem[operand1, e, esize];
   bits(esize) element2 = Elem[operand2, e, esize];
   if ElemP[mask, e, esize] == '1' then
      Elem[result, e, esize] = FPAbs(FPSub(element1, element2, FPCR[]));
   else
      Elem[result, e, esize] = element1;
Z[dn] = result;

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FABS

Floating-point absolute value (predicated).

Take the absolute value of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This clears the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | P  | g  | Z  | n  | Z  | d  |

FABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVE</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
   bits(esize) element = Elem(operand, e, esize);
   if ElemP[mask, e, esize] == '1' then
      ElemP[mask, e, esize] = '0';
      Elem[result, e, esize] = FPAbs(element);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FAC<cc>**

Floating-point absolute compare vectors.

Compare active absolute values of floating-point elements in the first source vector with corresponding absolute values of elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

The <cc> symbol specifies one of the standard ARM condition codes: GE, GT, LE, or LT.

This instruction is used by the pseudo-instructions FACLE, and FACLT.

It has encodings from 2 classes: Greater than and Greater than or equal

**Greater than**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 0 0 1 0 1 0 1 | Zm | 1 1 1 | Pg | Zn | 1 | Pd |

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;

**Greater than or equal**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 1 0 0 1 0 1 0 1 | Zm | 1 1 0 | Pg | Zn | 1 | Pd |

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;

**Assembler Symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(PL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    case op of
      when Cmp_GE res = FPCompareGE(FPAbs(element1), FPAbs(element2), FPCR[]);
      when Cmp_GT res = FPCompareGT(FPAbs(element1), FPAbs(element2), FPCR[]);
      else
        ElemP[result, e, esize] = if res then '1' else '0';
    end case
  else
    ElemP[result, e, esize] = '0';
end if
P[d] = result;
FACLE

Floating-point absolute compare less than or equal.

Compare active absolute values of floating-point elements in the first source vector being less than or equal to corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FAC<cc>. This means:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode for this instruction.

FACLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FAC<cc> gives the operational pseudocode for this instruction.
**FACLT**

Floating-point absolute compare less than.

Compare active absolute values of floating-point elements in the first source vector being less than corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FAC<cc>. This means:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

FACLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**Operation**

The description of FAC<cc> gives the operational pseudocode for this instruction.
FADD (immediate)

Floating-point add immediate (predicated).

Add an immediate to each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0') else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.5</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMax(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FADD (vectors, predicated)

Floating-point add vector (predicated).

Add active floating-point elements of the second source vector to corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |

FADD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

| <Zdn> | Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field. |
| <T>   | Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

| <Pg> | Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field. |
| <Zm> | Is the name of the second source scalable vector register, encoded in the "Zm" field. |

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bite(PL) mask = P[g];
bite(VL) operandl = Z[dn];
bite(VL) operando2 = Z[m];
bite(VL) result;
for e = 0 to elements - 1
  bite(esize) element1 = Elem[operandl, e, esize];
bite(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FADD (vectors, unpredicated)

Floating-point add vector (unpredicated).

Add all floating-point elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | size | 0   | Zm | 0  | 0  | 0  | 0  | 0  | Zn | Zd |

FADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1

    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

Z[d] = result;
FADDA

Floating-point add strictly-ordered reduction, accumulating in scalar.

Floating-point add a SIMD&FP scalar source and all active lanes of the vector source and place the result destructively in the SIMD&FP scalar source register. Vector elements are processed strictly in order from low to high, with the scalar source providing the initial value. Inactive elements in the source vector are ignored.

FADDA <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;V&gt;</th>
<th>Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<p>| &lt;dn&gt; | Is the number [0-31] of the source and destination SIMD&amp;FP register, encoded in the &quot;Vdn&quot; field. |
| &lt;Pg&gt; | Is the name of the governing scalable predicate register P0-P7, encoded in the &quot;Pg&quot; field. |
| &lt;Zm&gt; | Is the name of the source scalable vector register, encoded in the &quot;Zm&quot; field. |</p>
<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
<th>Is the size specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;T&gt;</td>
</tr>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(esize) operand1 = V[dn];
bits(VL) operand2 = Z[m];
bits(esize) result = operand1;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand2, e, esize];
        result = FPAdd(result, element, FPCR[{}]);
V[dn] = result;
FADD P

Floating-point add pairwise.

Add pairs of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

FADD P <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[dn];
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    if IsEven(e) then
      element1 = Elem[operand1, e + 0, esize];
      element2 = Elem[operand1, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
      element2 = Elem[operand2, e + 0, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);  
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FADDV

Floating-point add recursive reduction to scalar.

Floating-point add horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +0.0.

```
0 1 1 0 0 1 0 1 | size | 0 0 0 0 0 0 0 1 | Pg | Zn | Vd
```

FADDV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) identity = FPZero('0');
V[d] = ReducePredicated(ReduceOp_FADD, operand, mask, identity);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point complex add with rotate (predicated).

Add the real and imaginary components of the active floating-point complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±j beforehand. Destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

### Assembler Symbols

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<const>` Is the const specifier, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#90</td>
</tr>
<tr>
<td>1</td>
<td>#270</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for p = 0 to pairs-1
  acc_r  = Elem[operand1, 2 * p + 0, esize];
  acc_i  = Elem[operand1, 2 * p + 1, esize];
  elt2_r = Elem[operand2, 2 * p + 0, esize];
  elt2_i = Elem[operand2, 2 * p + 1, esize];
  if ElemP[mask, 2 * p + 0, esize] == '1' then
    if sub_i then elt2_i = FPNeg(elt2_i);
    acc_r = FPAdd(acc_r, elt2_i, FPCR[]);
  if ElemP[mask, 2 * p + 1, esize] == '1' then
    if sub_r then elt2_r = FPNeg(elt2_r);
    acc_i = FPAdd(acc_i, elt2_r, FPCR[]);
  Elem[result, 2 * p + 0, esize] = acc_r;
  Elem[result, 2 * p + 1, esize] = acc_i;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FCM<cc> (zero)**

Floating-point compare vector with zero.

Compare active floating-point elements in the source vector with zero, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

The <cc> symbol specifies one of the standard ARM condition codes: EQ, GE, GT, LE, LT, or NE.

It has encodings from 6 classes: Equal, Greater than, Greater than or equal, Less than, Less than or equal and Not equal.

### Equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 0 1 1 0 0 1 0 1 | size 0 1 0 0 1 0 0 0 1 | Pg | Zn | 0 | Pd |

\[
\text{eq}\;\text{lt}\;\text{ne}
\]

**FCMEQ** `<Pd>.<T>`, `<Pg>/Z`, `<Zn>.<T>` , #0.0

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEcmp op = Cmp_EQ;

### Greater than

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 0 1 1 0 0 1 0 1 | size 0 1 0 0 0 0 0 0 1 | Pg | Zn | 1 | Pd |

\[
\text{eq}\;\text{lt}\;\text{ne}
\]

**FCMGT** `<Pd>.<T>`, `<Pg>/Z`, `<Zn>.<T>` , #0.0

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEcmp op = Cmp_GT;

### Greater than or equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| 0 1 1 0 0 1 0 1 | size 0 1 0 0 0 0 0 0 1 | Pg | Zn | 0 | Pd |

\[
\text{eq}\;\text{lt}\;\text{ne}
\]

**FCMGE** `<Pd>.<T>`, `<Pg>/Z`, `<Zn>.<T>` , #0.0

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVEcmp op = Cmp_GE;
Less than

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 1 1 0 0 1 0 1 | size | 0 1 0 0 0 1 0 0 1 | Pg | Zn | 0 | Pd |

\[
\text{eq \ Cmp\_lt}\neq
\]

FCMLT <Pd>..<T>, <Pg>/Z, <Zn>..<T>, #0.0

\[
\text{if } !\text{HaveSVE() then UNDEFINED;}
\text{if size == '00' then UNDEFINED;}
\text{integer esize = 8 << UInt(size);}
\text{integer g = UInt(Pg);}
\text{integer n = UInt(Zn);}
\text{integer d = UInt(Pd);}
\text{SVECmp op = Cmp\_LT;}
\]

Less than or equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 1 1 0 0 1 0 1 | size | 0 1 0 0 0 1 0 0 1 | Pg | Zn | 1 | Pd |

\[
\text{eq \ Cmp\_le}\neq
\]

FCMLE <Pd>..<T>, <Pg>/Z, <Zn>..<T>, #0.0

\[
\text{if } !\text{HaveSVE() then UNDEFINED;}
\text{if size == '00' then UNDEFINED;}
\text{integer esize = 8 << UInt(size);}
\text{integer g = UInt(Pg);}
\text{integer n = UInt(Zn);}
\text{integer d = UInt(Pd);}
\text{SVECmp op = Cmp\_LE;}
\]

Not equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 1 1 0 0 1 0 1 | size | 0 1 0 0 1 1 0 0 1 | Pg | Zn | 0 | Pd |

\[
\text{eq \ Cmp\_ne}\neq
\]

FCMNE <Pd>..<T>, <Pg>/Z, <Zn>..<T>, #0.0

\[
\text{if } !\text{HaveSVE() then UNDEFINED;}
\text{if size == '00' then UNDEFINED;}
\text{integer esize = 8 << UInt(size);}
\text{integer g = UInt(Pg);}
\text{integer n = UInt(Zn);}
\text{integer d = UInt(Pd);}
\text{SVECmp op = Cmp\_NE;}
\]

Assembler Symbols

<\text{Pd}> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<\text{T}> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\text{T}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<\text{Pg}> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<\text{Zn}> Is the name of the source scalable vector register, encoded in the "Zn" field.
**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(PL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if Elem[mask, e, esize] == '1' then
        case op of
            when Cmp_EQ res = FPCompareEQ(element, 0<esize-1:0>, FPCR[]);
            when Cmp_GE res = FPCompareGE(element, 0<esize-1:0>, FPCR[]);
            when Cmp_GT res = FPCompareGT(element, 0<esize-1:0>, FPCR[]);
            when Cmp_NE res = FPCompareNE(element, 0<esize-1:0>, FPCR[]);
            when Cmp_LT res = FPCompareGT(0<esize-1:0>, element, FPCR[]);
            when Cmp_LE res = FPCompareGE(0<esize-1:0>, element, FPCR[]);
        Elem[result, e, esize] = if res then '1' else '0';
    else
        Elem[result, e, esize] = '0';

P[d] = result;
```

---

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FCM<cc> (vectors)**

Floating-point compare vectors.

Compare active floating-point elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags. The `<cc>` symbol specifies one of the standard ARM condition codes: EQ, GE, GT, or NE, with the addition of UO for an unordered comparison.

This instruction is used by the pseudo-instructions **FCMLE (vectors)**, and **FCMLT (vectors)**.

It has encodings from 5 classes: Equal, Greater than, Greater than or equal, Not equal and Unordered.

### Equal

```
0 1 1 0 0 1 0 1 size 0 Zm 0 1 1 Pg Zn 0 0 Pd
```

**FCMEQ** `<Pd>,<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>`

if `!HaveSVE()` then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_EQ;

### Greater than

```
0 1 1 0 0 1 0 1 size 0 Zm 0 1 0 Pg Zn 1 0 Pd
```

**FCMGt** `<Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>`

if `!HaveSVE()` then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEComp op = Cmp_GT;

### Greater than or equal

```
0 1 1 0 0 1 0 1 size 0 Zm 0 1 0 Pg Zn 0 0 Pd
```

**FCM<cc>** (vectors)
FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEcmp op = Cmp_GE;

Not equal

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | size | 0 | Zm | 0 | 1 | 1 | Pg | Zn | 1 | Pd |
| cmph | cmpl |

FCMNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEcmp op = Cmp_NE;

Unordered

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | size | 0 | Zm | 1 | 1 | 0 | Pg | Zn | 0 | Pd |

FCMUO <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVEcmp op = Cmp_UN;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

\begin{verbatim}
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(PL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        case op of
            when Cmp_EQ res = FPCompareEQ(element1, element2, FPCR[]);
            when Cmp_GE res = FPCompareGE(element1, element2, FPCR[]);
            when Cmp_GT res = FPCompareGT(element1, element2, FPCR[]);
            when Cmp_UN res = FPCompareUN(element1, element2, FPCR[]);
            when Cmp_LT res = FPCompareGT(element2, element1, FPCR[]);
            when Cmp_LE res = FPCompareGE(element2, element1, FPCR[]);
        ElemP[result, e, esize] = if res then '1' else '0';
        else
            ElemP[result, e, esize] = '0';

P[d] = result;
\end{verbatim}
**FCMLA (vectors)**

Floating-point complex multiply-add with rotate (predicated).

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element. Inactive elements in the destination vector register remain unmodified.

### Assembler Symbols

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the “Zm” field.
- `<const>` is the const specifier, encoded in “rot”:

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>
Operation

```plaintext
CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for p = 0 to pairs - 1
    addend_r = Elem[operand3, 2 * p + 0, esize];
    addend_i = Elem[operand3, 2 * p + 1, esize];
    elt1_a   = Elem[operand1, 2 * p + sel_a, esize];
    elt2_a   = Elem[operand2, 2 * p + sel_a, esize];
    elt2_b   = Elem[operand2, 2 * p + sel_b, esize];
    if ElemP[mask, 2 * p + 0, esize] == '1' then
        if neg_r then elt2_a = FPNeg(elt2_a);
        addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    if ElemP[mask, 2 * p + 1, esize] == '1' then
        if neg_i then elt2_b = FPNeg(elt2_b);
        addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCMLA (indexed)

Floating-point complex multiply-add by indexed values with rotate.

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation. Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding. These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

The complex numbers within the second source vector are specified using an immediate index which selects the same complex number position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the size of the complex number. This instruction is unpredicated.

It has encodings from 2 classes: Half-precision and Single-precision

**Half-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23</th>
<th>22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 1</td>
<td>0 1 i2</td>
</tr>
<tr>
<td>Zm 0 0 0 1</td>
<td>rot Zn Zda</td>
</tr>
</tbody>
</table>

```
```

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);

**Single-precision**

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23</th>
<th>22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 1</td>
<td>1 1 i1</td>
</tr>
<tr>
<td>Zm 0 0 0 1</td>
<td>rot Zn Zda</td>
</tr>
</tbody>
</table>

```
FCMLA <Zda>.S, <Zn>.S, <Zm>.S[iimm], <const>
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);

**Assembler Symbols**

`<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  For the half-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the single-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm>  For the half-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 3, encoded in the "i2" field.

For the single-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 1, encoded in the "i1" field.

<const>  Is the const specifier, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
integer pairpersegment = 128 DIV (2 * esize);
bids(VL) operand1 = Z[n];
bids(VL) operand2 = Z[m];
bids(VL) operand3 = Z[da];
bids(VL) result;

for p = 0 to pairs-1
    segmentbase = p - (p MOD pairpersegment);
    s = segmentbase + index;
    addend_r = Elem[operand3, 2 * p + 0, esize];
    addend_i = Elem[operand3, 2 * p + 1, esize];
    elt1_a = Elem[operand1, 2 * p + sel_a, esize];
    elt2_a = Elem[operand2, 2 * s + sel_a, esize];
    elt2_b = Elem[operand2, 2 * s + sel_b, esize];
    if neg_r then elt2_a = FPNeg(elt2_a);
    if neg_i then elt2_b = FPNeg(elt2_b);
    addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;

Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FCMLE (vectors)

Floating-point compare less than or equal to vector.

Compare active floating-point elements in the first source vector being less than or equal to corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FCM<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | Zm |
| 0   | 0  | cmph| cmpl|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

FCMLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T> is equivalent to

FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>.

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.
FCMLT (vectors)

Floating-point compare less than vector.

Compare active floating-point elements in the first source vector being less than corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FCMI<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCMI<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCMI<cc> (vectors) gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 1 1 0 0 1 0 1 | size | 0 | Zm | 0 | 1 | 0 | Pg | Zn | 1 | Pd
```

FCMLT <Pd>..<T>, <Pg>/Z, <Zm>..<T>, <Zn>..<T>

is equivalent to

FCMGT <Pd>..<T>, <Pg>/Z, <Zn>..<T>, <Zm>..<T>

Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FCMI<cc> (vectors) gives the operational pseudocode for this instruction.
Copy 8-bit floating-point immediate to vector elements (predicated).

Copy a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector remain unmodified.

This instruction is used by the alias FMOV (immediate, predicated).

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressable as ±n×16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) result = Z[d];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = imm;
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FCVT**

Floating-point convert precision (predicated).

Convert the size and precision of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

Since the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.


### Half-precision to single-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | Pg | Zn | Zd |

FCVT <Zd>.S, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;

### Half-precision to double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | Pg | Zn | Zd |

FCVT <Zd>.D, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;

### Single-precision to half-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | Pg | Zn | Zd |

FCVT <Zd>.H, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 16;
Single-precision to double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

FCVT <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 64;

Double-precision to half-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 1 1 0 0 1 0 0 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

FCVT <Zd>.H, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 16;

Double-precision to single-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

FCVT <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
  bits(esize) element = Elem[operand, e, esize];
  if ElemP[mask, e, esize] == '1' then
    bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR[]);
    Elem[result, e, esize] = ZeroExtend(res);
  end if

Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FCVTLT**

Floating-point up convert long (top, predicated).

Convert odd-numbered floating-point elements from the source vector to the next higher precision, and place the results in the active overlapping double-width elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 2 classes: Half-precision to single-precision and Single-precision to double-precision

### Half-precision to single-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 0 | 1 0 | 0 0 0 1 0 | 0 1 1 0 1 | Pg | Zn | Zd
```

FCVTLT <Zd>.S, <Pg>/M, <Zn>.H

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

### Single-precision to double-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 0 | 1 1 | 0 0 0 1 0 | 1 1 0 1 | Pg | Zn | Zd
```

FCVTLT <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

### Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

### Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        bits(esize DIV 2) element = Elem[operand, 2*e + 1, esize DIV 2];
        Elem[result, e, esize] = FPConvertSVE(element, FPCR[]);
    Z[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTNT

Floating-point down convert and narrow (top, predicated).

Convert active floating-point elements from the source vector to the next lower precision, and place the results in the odd-numbered half-width elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.

It has encodings from 2 classes: Single-precision to half-precision and Double-precision to single-precision

Single-precision to half-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  |
| Pg | Zn | Zd |

FCVTNT <Zd>.H, <Pg>/M, <Zn>.S

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Double-precision to single-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  |
| Pg | Zn | Zd |

FCVTNT <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, 2*e + 1, esize DIV 2] = FPCConvertSVE(element, FPCR[]);
Z[d] = result;
FCVTX

Floating-point down convert, rounding to odd (predicated).

Convert active double-precision floating-point elements from the source vector to single-precision, rounding to Odd, and place the results in the even-numbered 32-bit elements of the destination vector, while setting the odd-numbered elements to zero. Inactive elements in the destination vector register remain unmodified.

Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```plaintext
FCVTX <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;

integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR[], FPRounding_ODD);
        Elem[result, e, esize] = ZeroExtend(res);
Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FCVTXNT

Floating-point down convert, rounding to odd (top, predicated).

Convert active double-precision floating-point elements from the source vector to single-precision, rounding to Odd, and place the results in the odd-numbered 32-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified. Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

FCVTXNT <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, 2*e + 1, esize DIV 2] = FPConvertSVE(element, FPCR[], FPRounding_ODD);

Z[d] = result;
FCVTZS

Floating-point convert to signed integer, rounding toward zero (predicated).

Convert to the signed integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are sign-extended to fill each destination element.

It has encodings from 7 classes: Half-precision to 16-bit, Half-precision to 32-bit, Half-precision to 64-bit, Single-precision to 32-bit, Single-precision to 64-bit, Double-precision to 32-bit and Double-precision to 64-bit

Half-precision to 16-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 1 0 1 0 1 1 0 1 0 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1
         Pg Zn Zd
```

FCVTZS <Zd>.H, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Half-precision to 32-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 1 0 1 0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 0 1 0 1
         Pg Zn Zd
```

FCVTZS <Zd>.S, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Half-precision to 64-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 1 0 1
         Pg Zn Zd
```

FCVTZS
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 32-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1
int_U

FCVTZS <Zd>.S, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingZERO;

Single-precision to 64-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 1 1 0 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1
int_U

FCVTZS <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 32-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 1 1 0 1 1 0 0 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1 0 1
int_U

FCVTZS <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
Double-precision to 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |

\[
\text{int}_U \]

FCVTZS \(<\text{Zd}>.D, <\text{Pg}>/M, <\text{Zn}>.D\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s.esize = 64;
integer d.esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding\_ZERO;  

Assembler Symbols

\(<\text{Zd}>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.
\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\(<\text{Zn}>\) Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

\(\text{CheckSVEEnabled}()\);
integer elements = \(\text{VL} \div \text{esize}\);
bits(\(\text{PL}\)) mask = \(P[g]\);
bits(\(\text{VL}\)) operand = \(Z[n]\);
bits(\(\text{VL}\)) result = \(Z[d]\);

for \(e = 0\) to elements-1
   bits(esize) element = \(\text{Elem} \) \(\text{operand}, e, \text{esize}\);
   if \(\text{ElemP} \) \(\text{mask}, e, \text{esize} \) \(\approx \) '1' then
      bits(d.esize) res = \(\text{FPToFixed} \) \(\text{element}<s.esize-1:0>, 0, \text{unsigned}, \text{FPCR}[], \text{rounding}\);
      \(\text{Elem}\) result, e, esize] = \(\text{Extend} \)(res, unsigned);  
\(Z[d] = \text{result};\)

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FCVTZU**

Floating-point convert to unsigned integer, rounding toward zero (predicated).

Convert to the unsigned integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: Half-precision to 16-bit, Half-precision to 32-bit, Half-precision to 64-bit, Single-precision to 32-bit, Single-precision to 64-bit, Double-precision to 32-bit and Double-precision to 64-bit

### Half-precision to 16-bit

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |   |   |   |   | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Pg | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | Zn | Zd |

FCVTZU <Zd>.H, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

### Half-precision to 32-bit

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |   |   |   |   | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Pg | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | Zn | Zd |

FCVTZU <Zd>.S, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

### Half-precision to 64-bit

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |   |   |   |   | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Pg | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | Zn | Zd |

FCVTZU
FCVTZU <Zd>.D, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------------------|--|--|------------------|
| 0 1 1 0 0 1 0 1 1 0 0 1 1 0 1 | Pg | Zn | Zd |

FCVTZU <Zd>.S, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Single-precision to 64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------------------|--|--|------------------|
| 0 1 1 0 0 1 0 1 1 1 0 1 1 0 1 | Pg | Zn | Zd |

FCVTZU <Zd>.D, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;

Double-precision to 32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------------------------------|--|--|------------------|
| 0 1 1 0 0 1 0 1 1 1 0 1 1 0 1 0 | Pg | Zn | Zd |

FCVTZU <Zd>.S, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
Double-precision to 64-bit

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

\( \text{int}_U \)

FCVTZU \(<\text{Zd}>, <\text{Pg}>/\text{M}, <\text{Zn}>.\text{D} \)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding\_ZERO;

**Assembler Symbols**

\(<\text{Zd}>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.
\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\(<\text{Zn}>\) Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```
\text{CheckSVEEnabled}();
integer elements = \text{VL} \div \text{esize};
bits(\text{PL}) mask = P[g];
bits(\text{VL}) operand = Z[n];
bits(\text{VL}) result = Z[d];

\text{for } e = 0 \text{ to } \text{elements}-1 \\
\text{bits(}\text{esize}) \text{ element } = \text{Elem}(\text{operand, } e, \text{ esize});
\text{if } \text{ElemP[mask, } e, \text{ esize]} = '1' \text{ then }
\text{bits(}\text{d_esize}) \text{ res } = \text{FPToFixed}(\text{element}<\text{s_esize}-1:0>, 0, \text{unsigned, FPCR[]}, \text{rounding});
\text{Elem[result, } e, \text{ esize]} = \text{Extend}(\text{res, unsigned});

Z[d] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a \text{MOVPRFX} instruction. The \text{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \text{MOVPRFX} and this instruction is UNPREDICTABLE:

- The \text{MOVPRFX} instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The \text{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FDIV

Floating-point divide by vector (predicated).

Divide active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | Pg  | Zm  | Zdn |
```

FDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembrer Symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in "size":

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPDIV(element1, element2, FPCR[{}]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FDIVR

Floating-point reversed divide by vector (predicated).

Reversed divide active floating-point elements of the second source vector by corresponding floating-point elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 | size | 0 0 | 1 1 0 0 | 1 0 0 | Pg | Zm | Zdn
```

FDIVR <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>,<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPDiv(element2, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FDUP

Broadcast 8-bit floating-point immediate to vector elements (unpredicated).

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias FMOV (immediate, unpredicated).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>imm8</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>1 1 1 0 0 1 1 1 0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

FDUP <Zd>..<T>, #<const>

if !HaveSVE() then UNDEFINED;
if size == ’00’ then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Zd);
bits(esize) imm = VFPExpandImm(imm8);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a floating-point immediate value expressable as ±n/16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) result;

for e = 0 to elements-1
   Elem[result, e, esize] = imm;

Z[d] = result;
FEXPA

Floating-point exponential accelerator.

The FEXPA instruction accelerates the polynomial series calculation of the \( \exp(x) \) function.

The double-precision variant copies the low 52 bits of an entry from a hard-wired table of 64-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 11 bits of the source element (src<16:6>), setting the sign bit to zero.

The single-precision variant copies the low 23 bits of an entry from hard-wired table of 32-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 8 bits of the source element (src<13:6>), setting the sign bit to zero.

The half-precision variant copies the low 10 bits of an entry from hard-wired table of 16-bit coefficients, indexed by the low 5 bits of each element of the source vector, and prepends to that the next 5 bits of the source element (src<9:5>), setting the sign bit to zero.

A coefficient table entry with index \( M \) holds the floating-point value \( 2^{(m/64)} \), or for the half-precision variant \( 2^{(m/32)} \).

This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | size 1 | 0 0 0 0 0 | 1 0 1 1 1 0 | Zn | Zd
```

FEXPA \(<Zd>,<T>,<Zn>,<T>\)

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << Uint(size);
integer n = Uint(Zn);
integer d = Uint(Zd);

Assembler Symbols

- **\(<Zd>\)** Is the name of the destination scalable vector register, encoded in the “Zd” field.
- **\(<T>\)** Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- **\(<Zn>\)** Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPExpA(element);
Z[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FLOGB

Floating-point base 2 logarithm as integer.

This instruction returns the signed integer base 2 logarithm of each floating-point input element \(|x|\) after normalization.

This is the unbiased exponent of \(x\) used in the representation of the floating-point value, such that, for positive \(x\),

\[
x = \text{significand} \times 2^{\text{exponent}}.
\]

The integer results are placed in elements of the destination vector which have the same width (ESIZE) as the floating-point input elements:

* If \(x\) is normal, the result is the base 2 logarithm of \(x\).
* If \(x\) is subnormal, the result corresponds to the normalized representation.
* If \(x\) is infinite, the result is \(2^{(\text{esize}-1)}\).
* If \(x\) is ±0.0 or NaN, the result is \(-2^{(\text{esize}-1)}\).

Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 0 0 1 1</td>
</tr>
</tbody>
</table>

FLOGB <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn> Is the name of the source scalable vector register, encoded in the “Zn” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPLogB(element, FPCR[]);
Z[d] = result;
Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is \textbf{UNPREDICTABLE}:

- The \texttt{MOVPRFX} instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point fused multiply-add vectors (predicated), writing multiplicand \([Z_{dn} = Z_a + Z_{dn} \times Z_m]\).

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 0 1</td>
</tr>
</tbody>
</table>

FMAD \(<Z_{dn}>.<T>, <Pg>/M, <Zm>.<T>, <Z_a>.<T>\)

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

Assembler Symbols

\(<Z_{dn}>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

\(<Z_a>\) Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[a];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  bits(esize) element3 = Elem[operand3, e, esize];
  if ElemP[mask, e, esize] == '1' then
    if op1_neg then element1 = FPNeg(element1);  
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMAX (immediate)

Floating-point maximum with immediate (predicated).

Determine the maximum of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is NaN. Inactive elements in the destination vector remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 1 1 0 0 1 0 1 | size 0 1 1 1 0 1 0 0 | Pg 0 0 0 0 i1 | Zdn |

FMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

Assembler Symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T>    Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
      bits(esize) element1 = Elem[operand1, e, esize];
      if ElemP[mask, e, esize] == '1' then
          Elem[result, e, esize] = FPMax(element1, imm, FPCR[)];
      else
          Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMAX (vectors)**

Floating-point maximum (predicated).

Determine the maximum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If either element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

FMAX <Zdn>-.<T>, <Pg>/M, <Zdn>-.<T>, <Zm>-.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMAX(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMAXNM (immediate)**

Floating-point maximum number with immediate (predicated).

Determine the maximum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is the immediate. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMaxNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMAXNM (vectors)**

Floating-point maximum number (predicated).

Determine the maximum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is NaN then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  |

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAXNMP

Floating-point maximum number pairwise.

Compute the maximum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 1 1 0 0 1 0 0 | size | 0 1 0 1 0 0 1 0 0 | Pg | Zm | Zdn
```

FMAXNMP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[dn];
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    if IsEven(e) then
      element1 = Elem[operand1, e + 0, esize];
      element2 = Elem[operand1, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
      element2 = Elem[operand2, e + 0, esize];
    Elem[result, e, esize] = FPMAXNUM(element1, element2, FPCR[]);
  Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAXNMV

Floating-point maximum number recursive reduction to scalar.

Floating-point maximum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

FMAXNMV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) identity = FPDefaultNaN();

V[d] = ReducePredicated(ReduceOp_FMAXNUM, operand, mask, identity);
FMAXP

Floating-point maximum pairwise.

Compute the maximum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

![Register diagram]

### FMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

### Assembler Symbols

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

### Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[dn];
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
    Z[dn] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMAXV

Floating-point maximum recursive reduction to scalar.

Floating-point maximum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as -Infinity.

\[
\begin{array}{cccccccccccccccc}
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & Pg & Zn & Vd \\
\end{array}
\]

FMAXV \( <V><d>, <Pg>, <Zn> <T> \)

if \(!\text{HaveSVE}()\) then UNDEFINED;
if size == ‘00’ then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

\(<Zn>\) Is the name of the source scalable vector register, encoded in the “Zn” field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

\[ \text{CheckSVEEnabled}(); \]
\[ \text{bits}(\text{PL})\ \text{mask} = \ P[g]; \]
\[ \text{bits}(\text{VL})\ \text{operand} = \ Z[n]; \]
\[ \text{bits(esize)}\ \text{identity} = \ FPInfinity(‘1’); \]
\[ V[d] = \text{ReducePredicated}(\text{ReduceOp_FMAX}, \ \text{operand}, \ \text{mask}, \ \text{identity}); \]
FMIN (immediate)

Floating-point minimum with immediate (predicated).

Determine the minimum of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------------------------|-----------------|-----------------|
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | size | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | Pg | 0 | 0 | 0 | 0 | i1 | Zdn |

FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMin(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMIN (vectors)

Floating-point minimum (predicated).

Determine the minimum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If either element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 1 1 0 0 1 0 1 | size             | 0 0 0 1 1 1 1 0 0 | Pg               | Zm               | Zdn              |

FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P[]) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMIN(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMINNM (immediate)**

Floating-point minimum number with immediate (predicated).

Determine the minimum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is the immediate. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | Zdn |

FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros() else FPOne('0');

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMinNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMINNM (vectors)

Floating-point minimum number (predicated).

Determine the minimum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is NaN then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 1 1 0 0 1 0 1 size 0 0 0 1 0 1 1 0 0 Pg Zm Zdn
```

FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMInNum(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point minimum number pairwise.

Compute the minimum value of each pair of adjacent floating-point elements within each source vector, and interleave
the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.
NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet
NaN, the result is the numerical value.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| size            | Pg              | Zm              |
| 0 1 1 0 0 1 0 0 0 1 1 0 0 | <Zdn><T>, <Pg>/M, <Zdn><T>, <Zm><T> |

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[dn];
bits(esize) element1;
bits(esize) element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    if IsEven(e) then
      element1 = Elem[operand1, e + 0, esize];
      element2 = Elem[operand1, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
      element2 = Elem[operand2, e + 0, esize];
    Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
  Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMINNMV**

Floating-point minimum number recursive reduction to scalar.

Floating-point minimum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|
| 0 1 1 0 0 1 0 1 | size | 0 0 0 | 1 0 1 0 | 0 1 | Pg | Zn | Vd |

**FMINNMV** `<V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

**Assembler Symbols**

- `<V>`: Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;V&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<d>`: Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Zn>`: Is the name of the source scalable vector register, encoded in the "Zn" field.

- `<T>`: Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) identity = FPDefaultNaN();

V[d] = ReducePredicated(ReduceOp_FMINNUM, operand, mask, identity);

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMINP

Floating-point minimum pairwise.

Compute the minimum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[dn];
bids(esize) element1;
bids(esize) element2;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    if IsEven(e) then
      element1 = Elem[operand1, e + 0, esize];
      element2 = Elem[operand1, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
      element2 = Elem[operand2, e + 0, esize];
    Elem[result, e, esize] = FPMIN(element1, element2, FPCR[]);
  Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMINV

Floating-point minimum recursive reduction to scalar.

Floating-point minimum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +Infinity.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  |

FMINV <V><d>, <Pg>, <Zn>,<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

< Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(esize) identity = FPInfinity('0');

V[d] = ReducePredicated(ReduceOp_FMIN, operand, mask, identity);
FMLA (vectors)

Floating-point fused multiply-add vectors (predicated), writing addend \([Z_{da} = Z_{da} + Z_n \times Z_m]\).

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

Assembler Symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if ElemP[mask, e, esize] == '1' then
        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element3;
Z[da] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMLA (indexed)**

Floating-point fused multiply-add by indexed elements \( (Zda = Zda + Zn \times Zm[\text{indexed}] \)).

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added without intermediate rounding to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: [Half-precision](#), [Single-precision](#) and [Double-precision](#)

### Half-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 3h | 1  | i3l| Zm | 0  | 0  | 0  | 0  | 0  | 0  | Zn | Zda|
| op |


if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

### Single-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | i2 | Zm | 0  | 0  | 0  | 0  | 0  | 0  | Zn | Zda|
| size<1>size<0> | op |

FMLA \(<Zda>.S, <Zn>.S, <Zm>[<imm>]\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;

### Double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | i1 | Zm | 0  | 0  | 0  | 0  | 0  | 0  | Zn | Zda|
| size<1>size<0> | op |

FMLA \(<Zda>.D, <Zn>.D, <Zm>[<imm>]\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.
For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicate.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLALB (vectors)

Half-precision floating-point multiply-add long to single-precision (bottom).

This half-precision floating-point multiply-add long instruction widens the even-numbered 16-bit half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & Zm & 1 & 0 & 0 & 0 & 0 & 0 & Zn & Zda \\
\end{array}
\]


if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 0, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLALB (indexed)

Half-precision floating-point multiply-add long to single-precision (bottom, indexed).

This half-precision floating-point multiply-add long instruction widens the even-numbered 16-bit half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | i3h | Zm | 0  | 1  | 0  | 0  | i3l | 0  | Zn | Zda |


if !HaveSVE2() then UNDEFINED;
integer esize = 32;
n = UInt(Zn);
m = UInt(Zm);
da = UInt(Zda);
index = UInt(i3h:i3l);
op1_neg = FALSE;

Assembler Symbols

<Zda> is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm> is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
eltspersegment = 128 DIV esize;
operand1 = Z[n];
operand2 = Z[m];
operand3 = Z[da];
result;
for e = 0 to elements-1
    segmentbase = e - (e MOD eltspersegment);
    s = 2 * segmentbase + index;
    element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    element2 = Elem[operand2, s, esize DIV 2];
    element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    result, e, esize = FPMulAddH(element3, element1, element2, FPCR[]);

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLALT (vectors)

Half-precision floating-point multiply-add long to single-precision (top).

This half-precision floating-point multiply-add long instruction widens the odd-numbered 16-bit half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 0 1 0 1 Zm 1 0 0 0 0 1 Zn Zda
```


if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 1, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMLALT (indexed)**

Half-precision floating-point multiply-add long to single-precision (top, indexed).

This half-precision floating-point multiply-add long instruction widens the odd-numbered 16-bit half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
0  1  1  0  0  1  0  0  1  0  1  i3h  Zm  0  1  0  0  i3l  1  Zn  Zda
``` 

FMLALT <Zda>.S, <Zn>.H, <Zm>.H[i3]

```
if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = FALSE;
```

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register; encoded in the “Zn” field.
- `<Zm>` Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
- `<imm>` Is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
### FMLS (vectors)

Floating-point fused multiply-subtract vectors (predicated), writing addend \([Zda = Zda + -Zn \times Zm]\).

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | Zm | 0  | 0  | 1  | Pg | 0  | 0  | 1  | Zn | 0  | Zda| N  | op |

**FMLS** <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

**Assembler Symbols**

- **<Zda>**: Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- **<T>**: Is the size specifier, encoded in “size”:
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- **<Pg>**: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>**: Is the name of the first source scalable vector register, encoded in the “Zn” field.
- **<Zm>**: Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  bits(esize) element3 = Elem[operand3, e, esize];
  if ElemP[mask, e, esize] == '1' then
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element3;
Z[da] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLS (indexed)

Floating-point fused multiply-subtract by indexed elements (Zda = Zda + -Zn * Zm[i]).

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted without intermediate rounding from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision and Double-precision

Half-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>op</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

Single-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
</tr>
</tbody>
</table>

FMLS <Zda>.S, <Zn>.S, <Zm>[<imm>]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

Double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 0 1 0 1 0</td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
</tr>
</tbody>
</table>

FMLS <Zda>.D, <Zn>.D, <Zm>[<imm>]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the “i2” field.

For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the “i1” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);

Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLSLB (vectors)

Half-precision floating-point multiply-subtract long from single-precision (bottom).

This half-precision floating-point multiply-subtract long instruction widens the even-numbered 16-bit half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

```
0 1 1 0 0 1 0 1 1 0 0 1 0 0 0 0
Zm 1 0 1 0 0 0 0 0
Zn
Zda
```


if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 0, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLSLB (indexed)

Half-precision floating-point multiply-subtract long from single-precision (bottom, indexed).

This half-precision floating-point multiply-subtract long instruction widens the even-numbered 16-bit half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 0 1 0 1 0 1 i3h i3l Zm 0 1 1 0 i3l 0 Zn Zda
```


if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLSLT (vectors)

Half-precision floating-point multiply-subtract long from single-precision (top).

This half-precision floating-point multiply-subtract long instruction widens the odd-numbered 16-bit half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|-----------------|----------------|
| 0 1 1 0 0 1 0 0 1 0 1 | Zm | 1 0 1 0 0 1 | Zn | Zda |
| o2 | op | T |


if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
  bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 1, esize DIV 2];
  bits(esize) element3 = Elem[operand3, e, esize];
  if op1_neg then element1 = FPNeg(element1);
  Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMLSLT (indexed)

Half-precision floating-point multiply-subtract long from single-precision (top, indexed).

This half-precision floating-point multiply-subtract long instruction widens the odd-numbered 16-bit half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the overlapping 32-bit single-precision elements of the addend and destination vector. This instruction is unpredicated.


generate


if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm> Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMMLA

Floating point matrix multiply-accumulate.

The floating point matrix multiply-accumulate instruction supports single-precision and double-precision data types in a 2×2 matrix contained in segments of 128 or 256 bits, respectively. It multiplies the 2×2 matrix in each segment of the first source vector by the 2×2 matrix in the corresponding segment of the second source vector. The resulting 2×2 matrix product is then destructively added to the matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 2-way dot product per destination element. This instruction is unpredicated. The single-precision variant is vector length agnostic. The double-precision variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F32MM indicates whether the single-precision variant is implemented.
ID_AA64ZFR0_EL1.F64MM indicates whether the double-precision variant is implemented.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Zn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMMLA <Zda>.S, <Zn>.S, <Zm>.S

if !HaveSVEFP32MatMulExt() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

### 64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Zn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FMMLA <Zda>.D, <Zn>.D, <Zm>.D

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

### Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.
Operation

CheckSVEEnabled();
if \( VL < \text{esize} \times 4 \) then UNDEFINED;
integer segments = \( VL \div (4 \times \text{esize}) \);
bits(\( VL \)) operand1 = \( Z[n] \);
bits(\( VL \)) operand2 = \( Z[m] \);
bits(\( VL \)) operand3 = \( Z[da] \);
bits(\( 4\times\text{esize} \)) result = \( \text{Zeros}() \);
bits(\( 4\times\text{esize} \)) op1, op2;
bits(\( 4\times\text{esize} \)) res, addend;

for \( s = 0 \) to segments-1
    op1 = \( \text{Elem}[\text{operand1}, s, 4\times\text{esize}] \);
    op2 = \( \text{Elem}[\text{operand2}, s, 4\times\text{esize}] \);
    addend = \( \text{Elem}[\text{operand3}, s, 4\times\text{esize}] \);
    res = \( \text{FPMatMulAdd}(\text{addend}, \text{op1}, \text{op2}, \text{esize}, \text{FPCR[]}) \);
    \( \text{Elem}[\text{result}, s, 4\times\text{esize}] = \text{res} \);
\( Z[da] = \text{result} \);

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMOV (zero, predicated)

Move floating-point +0.0 to vector elements (predicated).

Move floating-point constant +0.0 to to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is a pseudo-instruction of **CPY (immediate, merging)**. This means:

- The encodings in this description are named to match the encodings of **CPY (immediate, merging)**.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of **CPY (immediate, merging)** gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 0 0 0 0 1 0 1 | size | 0 1 | Pg | 0 1 | 0 0 0 0 0 0 0 0 | Zd |
|-----------------|------|-----|----|----|----|-----|-----|
| M sh            | imm8 |

FMOV <Zd>.<T>, <Pg>/M, #0.0

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #0

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0 1</td>
<td>H</td>
</tr>
<tr>
<td>1 0</td>
<td>S</td>
</tr>
<tr>
<td>1 1</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register, encoded in the "Pg" field.

**Operation**

The description of **CPY (immediate, merging)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FMOV (zero, unpredicated)

Move floating-point +0.0 to vector elements (unpredicated).

Unconditionally broadcast the floating-point constant +0.0 into each element of the destination vector. This instruction is unpredicated.

This is a pseudo-instruction of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of DUP (immediate) gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Zd |
| sh  | imm8 |

FMOV <Zd>.<T>, #0.0

is equivalent to

DUP <Zd>.<T>, #0

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
FMOV (immediate, predicated)

Move 8-bit floating-point immediate to vector elements (predicated).

Move a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of FCPY. This means:

- The encodings in this description are named to match the encodings of FCPY.
- The description of FCPY gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 size 0 1 Pg 1 1 0 imm8 Zd

FMOV <Zd>, <T>, <Pg>/M, #<const>

is equivalent to

FCPY <Zd>, <T>, <Pg>/M, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<const> Is a floating-point immediate value expressable as ±n+16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of FCPY gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMOV (immediate, unpredicated)**

Move 8-bit floating-point immediate to vector elements (unpredicated).

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This is an alias of **FDUP**. This means:

- The encodings in this description are named to match the encodings of **FDUP**.
- The description of **FDUP** gives the operational pseudocode for this instruction.

![Encoding Table]

FMOV `<Zd>`, `<T>`, #<const>

is equivalent to

FDUP `<Zd>`, `<T>`, #<const>

and is always the preferred disassembly.

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a floating-point immediate value expressable as ±n×16^-r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the “imm8” field.

**Operation**

The description of **FDUP** gives the operational pseudocode for this instruction.
FMSB

Floating-point fused multiply-subtract vectors (predicated), writing multiplicand \(Z_{dn} = Z_a + Z_{dn} \cdot Z_m\).

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

FMSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

\texttt{CheckSVEEnabled();}
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[a];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = \texttt{Elem}[operand1, e, esize];
    bits(esize) element2 = \texttt{Elem}[operand2, e, esize];
    bits(esize) element3 = \texttt{Elem}[operand3, e, esize];

    if \texttt{ElemP}[mask, e, esize] == '1' then
        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        \texttt{Elem}[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        \texttt{Elem}[result, e, esize] = element1;

Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMUL (immediate)**

Floating-point multiply by immediate (predicated).

Multiply by an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +2.0 only. Inactive elements in the destination vector register remain unmodified.

![Register file](image)

FMUL $Zdn$.<$T>$, <$Pg>/M, <$Zdn$.<$T>$, <$const$>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0') else FPTwo('0');

**Assembler Symbols**

$<Zdn>$ Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

$<T>$ Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$<Pg>$ Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

$<const>$ Is the floating-point immediate value, encoded in "i1":

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.5</td>
</tr>
<tr>
<td>1</td>
<td>#2.0</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = $VL$ DIV esize;
bits($PL$) mask = $P[g]$;
bits($VL$) operand1 = $Z[dn]$;
bits($VL$) result;
for e = 0 to elements-1
  bits(esize) element1 = $Elem[operand1, e, esize]$;
  if $ElemP[mask, e, esize] == '1' then
    $Elem[result, e, esize] = FPMul(element1, imm, FPCR[])$;
  else
    $Elem[result, e, esize] = element1$;
$Z[dn] = result$;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMUL (vectors, predicated)**

Floating-point multiply vectors (predicated).

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**FMUL** <Zdn>.<<T>, <Pg>/M, <Zm>.<<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

< Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FMUL (vectors, unpredicated)**

Floating-point multiply vectors (unpredicated).

Multiply all elements of the first source vector by corresponding floating-point elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |
```

**FMUL** `<Zd>`.<T>, `<Zn>`.<T>, `<Zm>`.<T>

if `!HaveSVE()` then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the “Zd” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
Z[d] = result;
```
**FMUL (indexed)**

Floating-point multiply by indexed elements.

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: Half-precision, Single-precision and Double-precision.

### Half-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | i3h| i3l| Zm | 0  | 0  | 1  | 0  | 0  | 0  | Zn | Zd |


if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

### Single-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | i2 | Zm | 0  | 0  | 1  | 0  | 0  | 0  | Zn | Zd |

FMUL <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

### Double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | i1 | Zm | 0  | 0  | 1  | 0  | 0  | 0  | Zn | Zd |


if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

### Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.

For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, s, esize];
  Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
Z[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FMULX**

Floating-point multiply-extended vectors (predicated).

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector except that \( \infty \times 0.0 \) gives 2.0 instead of NaN, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

The instruction can be used with FRECPX to safely convert arbitrary elements in mathematical vector space to UNIT VECTORS or DIRECTION VECTORS with length 1.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 1 1 0 0 1 0 1 | size | 0 0 1 0 1 0 1 0 0 | Pg | Zm | Zdn |

FMULX \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
if size == \('00'\) then UNDEFINED;
integer esize = 8 << \(\text{UInt}(size)\);
integer g = \(\text{UInt}(Pg)\);
integer dn = \(\text{UInt}(Zdn)\);
integer m = \(\text{UInt}(Zm)\);

**Assembler Symbols**

\(<Zdn>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```asm
CheckSVEEnabled();
integer elements = \(\text{VL} / \text{esize}\);
bits(\(\text{PL}\)) mask = \(\text{P}[g]\);
bits(\(\text{VL}\)) operand1 = \(\text{Z}[dn]\);
bits(\(\text{VL}\)) operand2 = \(\text{Z}[m]\);
bits(\(\text{VL}\)) result;
for e = 0 to elements-1
  bits(esize) element1 = \(\text{Elem}[\text{operand1}, e, \text{esize}]\);
  bits(esize) element2 = \(\text{Elem}[\text{operand2}, e, \text{esize}]\);
  if \(\text{Elem}[\text{mask}, e, \text{esize}] == '1'\) then
    \(\text{Elem}[\text{result}, e, \text{esize}] = \text{FPMULX}(\text{element1, element2, FPCR[\()]};\)
  else
    \(\text{Elem}[\text{result}, e, \text{esize}] = \text{element1};\)
\(\text{Z}[dn] = \text{result};\)
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FNEG

Floating-point negate (predicated).

Negate each active floating-point element of the source vector, and place the results in the corresponding elements of
the destination vector. This inverts the sign bit and cannot signal a floating-point exception. Inactive elements in the
destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 1 0 0 | size | 0 1 1 1 0 1 1 0 1 | Pg | Zn | Zd |

FNEG <Zd>.<T>, <Pg>./M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(esize) element = Elem{operand, e, esize};
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPNeg(element);

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FNMAD

Floating-point negated fused multiply-add vectors (predicated), writing multiplicand \[Zdn = -Za + -Zdn * Zm\].

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 1 1 0 0 1 0 1 | size | 1 | Za | 1 | 1 | 0 | Pg | Zm | Zdn |

N op

FNMAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[a];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if ElemP[mask, e, esize] == '1' then
        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point negated fused multiply-add vectors (predicated), writing addend \([\text{Zda} = -\text{Zda} + -\text{Zn} \times \text{Zm}]\).

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

---

**Assembler Symbols**

\(<\text{Zda}>\) Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

\(<\text{T}>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zn}>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

\(\text{CheckSVEEnabled}()\);

integer elements = \(\text{VL} \div\) esize;
bits(PL) mask = \(P[g]\);
bits(VL) operand1 = \(Z[n]\);
bits(VL) operand2 = \(Z[m]\);
bits(VL) operand3 = \(Z[da]\);
bits(VL) result;
for \(e = 0\) to elements-1

bits(esize) element1 = \(\text{Elem}[\text{operand1}, e, \text{esize}]\);
bits(esize) element2 = \(\text{Elem}[\text{operand2}, e, \text{esize}]\);
bits(esize) element3 = \(\text{Elem}[\text{operand3}, e, \text{esize}]\);

if \(\text{Elem}[\text{mask}, e, \text{esize}] == '1'\) then

if op1_neg then element1 = \(\text{FPNeg}(\text{element1})\);
if op3_neg then element3 = \(\text{FPNeg}(\text{element3})\);
\(\text{Elem}[\text{result}, e, \text{esize}] = \text{FPMulAdd}(\text{element3}, \text{element1}, \text{element2}, \text{FPCR[]}\));
else

\(\text{Elem}[\text{result}, e, \text{esize}] = \text{element3}\);

\(Z[\text{da}] = \text{result}\);
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FNMLS

Floating-point negated fused multiply-subtract vectors (predicated), writing addend [Zda = -Zda + Zn * Zm].

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

FNMLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer n = Uint(Zn);
integer m = Uint(Zm);
integer da = Uint(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    bits(esize) element3 = Elem[operand3, e, esize];
    if ElemP[mask, e, esize] == '1' then
        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    end
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand \([Z_{dn} = -Z_a + Z_{dn} \times Z_m]\).

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | \begin{array}{c|c|c|c} \text{size} & 1 & Za & 11 \\ \hline \text{Pg} & 1 & Zm & \end{array} |

---

**Assembler Symbols**

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<Za>** Is the name of the third source scalable vector register, encoded in the "Za" field.

**Operation**

\[ \text{CheckSVEEnabled();} \]
\[ \text{integer elements = VL DIV esize;} \]
\[ \text{bits(PL) mask = P[g];} \]
\[ \text{bits(VL) operand1 = Z[dn];} \]
\[ \text{bits(VL) operand2 = Z[m];} \]
\[ \text{bits(VL) operand3 = Z[a];} \]
\[ \text{bits(VL) result;} \]

for \( e = 0 \) to \( \text{elements-1} \)
\[ \text{bits(size) element1 = Elem[operand1, e, esize];} \]
\[ \text{bits(size) element2 = Elem[operand2, e, esize];} \]
\[ \text{bits(size) element3 = Elem[operand3, e, esize];} \]

\[ \text{if ElemP[mask, e, esize] == '1' then} \]
\[ \text{if op1_neg then element1 = FPNeg(element1);} \]
\[ \text{if op3_neg then element3 = FPNeg(element3);} \]
\[ \text{Elem[result, e, esize] = FP_MUL_ADD(element3, element1, element2, FPCR[]);} \]
\[ \text{else} \]
\[ \text{Elem[result, e, esize] = element1;} \]
\[ Z[dn] = result; \]
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FRECPE

Floating-point reciprocal estimate (unpredicated).

Find the approximate reciprocal of each floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

FRECPE <Zd>..<T>, <Zn>..<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand = Z[n];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRrecipEstimate(element, FPCR[]);
Z[d] = result;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FRECPS**

Floating-point reciprocal step (unpredicated).

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 2.0 without intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal of a vector of floating-point values.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Zm</th>
<th></th>
<th></th>
<th>Zn</th>
<th></th>
<th></th>
<th>Zd</th>
</tr>
</thead>
</table>
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0

**FRECPS** <Zd>..<T>, <Zn>.<T>, <Zm>.<T>

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler Symbols**

<Zd>  Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T>  Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm>  Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);
Z[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**FRECPX**

Floating-point reciprocal exponent (predicated).

Invert the exponent and zero the fractional part of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector.Inactive elements in the destination vector register remain unmodified.

The result of this instruction can be used with F_MULX to convert arbitrary elements in mathematical vector space to "unit vectors" or "direction vectors" of length 1.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FRECPX `<Zd>,<T>, <Pg>/M, <Zn>`.

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
  bits(esize) element = Elem[operand, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPRecpX(element, FPCR[]);
Z[d] = result;

Operational Information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point round to integral value (predicated).

Round to an integral floating-point value with the specified rounding option from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

The <r> symbol specifies one of the following rounding options: N (to nearest, with ties to even), A (to nearest, with ties away from zero), M (toward minus Infinity), P (toward plus Infinity), Z (toward zero), I (current FPCR rounding mode), or X (current FPCR rounding mode, signalling inexact).

It has encodings from 7 classes: **Current mode**, **Current mode signalling inexact**, **Nearest with ties to away**, **Nearest with ties to even**, **Toward zero**, **Toward minus infinity** and **Toward plus infinity**

### Current mode

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | size | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | Pg  | Zn  | Zd  |

**FRINTI** \(<Zd>.<T>, <Pg>/M, <Zn>.<T>\)

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

### Current mode signalling inexact

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | size | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | Pg  | Zn  | Zd  |

**FRINTX** \(<Zd>.<T>, <Pg>/M, <Zn>.<T>\)

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

### Nearest with ties to away

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | size | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | Pg  | Zn  | Zd  |

**FRINTA** \(<Zd>.<T>, <Pg>/M, <Zn>.<T>\)

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEAWAY;
Nearest with ties to even

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 size 0 0 0 0 0 1 0 1 Pg Zn Zd

FRINTN <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEEVEN;

Toward zero

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 size 0 0 0 0 1 1 1 0 1 Pg Zn Zd

FRINTZ <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_ZERO;

Toward minus infinity

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 size 0 0 0 0 1 0 1 0 1 Pg Zn Zd

FRINTM <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_NEGINF;

Toward plus infinity

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 size 0 0 0 0 1 1 0 1 Pg Zn Zd

FRINTP <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_POSINF;
Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point reciprocal square root estimate (unpredicated).

Find the approximate reciprocal square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

### Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

### Operation

1. `CheckSVEEnabled();`
2. `integer elements = VL DIV esize;`
3. `bits(VL) operand = Z[n];`
4. `bits(VL) result;`
5. For `e = 0` to `elements-1`:
   - `bits(esize) element = Elem[operand, e, esize];`
   - `Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR[]);`
6. `Z[d] = result;`
**FRSQRTS**

Floating-point reciprocal square root step (unpredicated).

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 3.0 and divide the results by 2.0 without any intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal square root of a vector of floating-point values.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 | size | 0 | Zm | 0 0 0 1 1 1 | Zn | Zd
```

**FRSQRTS** <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if ![HaveSVE()] then UNDEFINED;
if size == ‘00’ then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bis(VL) operand1 = Z[n];
bis(VL) operand2 = Z[m];
bis(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
bis(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPRsqrtStepFused(element1, element2);
Z[d] = result;
**FScale**

Floating-point adjust exponent by vector (predicated).

Multiply the active floating-point elements of the first source vector by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 1 1 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 1 |

**FScale** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  integer element2 = SInt(Elem[operand2, e, esize]);
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPScale(element1, element2, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Floating-point square root (predicated).

Calculate the square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 1 1 0 0 1 0 1 | 0 0 1 1 0 1 1 0 1 | Pg | Zn | Zd |
```

FSQRT <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem(operand, e, esize);
    if Elem(mask, e, esize) == '1' then
        Elem(result, e, esize) = FPSqrt(element, FPCR[]);
Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUB (immediate)

Floating-point subtract immediate (predicated).

Subtract an immediate from each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 size 0 1 1 0 0 1 1 0 0 Pg 0 0 0 0 i1 Zdn
```

FSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPointFive('0') else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.5</td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPSub(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FSUB (vectors, predicated)**

Floating-point subtract vectors (predicated).

Subtract active floating-point elements of the second source vector from corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

FSUB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Asmmler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUB (vectors, unpredicated)

Floating-point subtract vectors (unpredicated).

Subtract all floating-point elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

FSUB <Zd>,<T>, <Zn>,<T>, <Zm>,<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FSUBR (immediate)

Floating-point reversed subtract from immediate (predicated).

Reversed subtract from an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |

FSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0') else FPOne('0');

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in "i1":

<table>
<thead>
<tr>
<th>i1</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.5</td>
</tr>
<tr>
<td>0</td>
<td>#1.0</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPSub(imm, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FSUBR (vectors)

Floating-point reversed subtract vectors (predicated).

Reversed subtract active floating-point elements of the first source vector from corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Pg</td>
<td>Zm</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FSUBR <Zdn>,<T>, <Pg>/M, <Zdn>..<T>, <Zm>..<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = FPSub(element2, element1, FPCR[]);
  else
    Elem[result, e, esize] = element1;
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
FTMAD

Floating-point trigonometric multiply-add coefficient.

The FTMAD instruction calculates the series terms for either \( \sin(x) \) or \( \cos(x) \), where the argument \( x \) has been adjusted to be in the range \( \frac{-\pi}{4} < x \leq \frac{\pi}{4} \).

To calculate the series terms of \( \sin(x) \) and \( \cos(x) \) the initial source operands of FTMAD should be zero in the first source vector and \( x^2 \) in the second source vector. The FTMAD instruction is then executed eight times to calculate the sum of eight series terms, which gives a result of sufficient precision.

The FTMAD instruction multiplies each element of the first source vector by the absolute value of the corresponding element of the second source vector and performs a fused addition of each product with a value obtained from a table of hard-wired coefficients, and places the results destructively in the first source vector.

The coefficients are different for \( \sin(x) \) and \( \cos(x) \), and are selected by a combination of the sign bit in the second source element and an immediate index in the range 0 to 7.

This instruction is unpredicated.

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

<imm> Is the unsign immediate operand, in the range 0 to 7, encoded in the “imm3” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = FPTrigMAdd(imm, element1, element2, FPCR[]);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**FTSMUL**

Floating-point trigonometric starting value.

The FTSMUL instruction calculates the initial value for the FTMAD instruction. The instruction squares each element in the first source vector and then sets the sign bit to a copy of bit 0 of the corresponding element in the second source register, and places the results in the destination vector. This instruction is unpredicated.

To compute \( \sin(x) \) or \( \cos(x) \) the instruction is executed with elements of the first source vector set to \( x \), adjusted to be in the range \(-\pi/4 < x \leq \pi/4\).

The elements of the second source vector hold the corresponding value of the quadrant \( q \) number as an integer not a floating-point value. The value \( q \) satisfies the relationship \((2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4\).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
FTSMUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the “Zd” field.
- `<T>`: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>`: Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigSMul(element1, element2, FPCR[]);
Z[d] = result;
```
FTSSEL

Floating-point trigonometric select coefficient.

The FTSSEL instruction selects the coefficient for the final multiplication in the polynomial series approximation. The instruction places the value 1.0 or a copy of the first source vector element in the destination element, depending on bit 0 of the quadrant number \(q\) held in the corresponding element of the second source vector. The sign bit of the destination element is copied from bit 1 of the corresponding value of \(q\). This instruction is unpredicated.

To compute \(\sin(x)\) or \(\cos(x)\) the instruction is executed with elements of the first source vector set to \(x\), adjusted to be in the range \(-\pi/4 < x \leq \pi/4\).

The elements of the second source vector hold the corresponding value of the quadrant \(q\) number as an integer not a floating-point value. The value \(q\) satisfies the relationship \((2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4\).

\[
\begin{array}{ccccccccccccccccccccccccccc}
\end{array}
\]

| 0 | 0 | 0 | 0 | 1 | 0 | 0 | size | 1 | Zm | 1 | 0 | 1 | 1 | 0 | 0 | Zn | Zd |

FTSSEL <Zd><T>, <Zn><T>, <Zm><T>

if !HaveSVE() then UNDEFINED;
if size == ‘00’ then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigSSel(element1, element2);
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
HISTCNT

Count matching elements in vector.

This instruction compares each active 32 or 64-bit element of the first source vector with all active elements with an element number less than or equal to its own in the second source vector, and places the count of matching elements in the corresponding element of the destination vector.Inactive elements in the destination vector are set to zero.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

HISTCNT <Zd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
integer n = UInt(Zn);
integer m = UInt(Zm);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer count = 0;
    if Elem[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        for i = 0 to e
            if Elem[mask, i, esize] == '1' then
                bits(esize) element2 = Elem[operand2, i, esize];
                if element1 == element2 then
                    count = count + 1;

        Elem[result, e, esize] = count<esize-1:0>;

Z[d] = result;
HISTSEG

Count matching elements in vector segments.

This instruction compares each 8-bit byte element of the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector and places the count of matching elements in the corresponding element of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 0 0 1 0 1 | size | 1 | Zm | 1 0 1 0 0 0 | Zn | Zd |

HISTSEG <Zd>.B, <Zn>.B, <Zm>.B

if !HaveSVE2() then UNDEFINED;
if size != '00' then UNDEFINED;
integer esize = 8;
integer d = UInt(Zd);
integer n = UInt(Zn);
integer m = UInt(Zm);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer segments = VL DIV 128;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for b = 0 to segments-1
  for s = 0 to eltspersegment-1
    integer count = 0;
    integer e = eltspersegment * b + s;
    bits(esize) element1 = Elem[operand1, e, esize];
    for i = 0 to eltspersegment-1
      integer e2 = eltspersegment * b + i;
      bits(esize) element2 = Elem[operand2, e2, esize];
      if element1 == element2 then
        count = count + 1;
      Elem[result, e, esize] = count<esize-1:0>;
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
INCB, INCD, INCH, INCW (scalar)

Increment scalar by multiple of predicate constraint element count.

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte, Doubleword, Halfword and Word

**Byte**

```
31 30 29 28 27 26 25 24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  1  0  0  0  0  1  1  imm4  1  1  1  0  0  0  0  pattern  Rdn
```

```
INCB <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

**Doubleword**

```
31 30 29 28 27 26 25 24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  1  0  0  1  1  1  1  imm4  1  1  1  0  0  0  0  pattern  Rdn
```

```
INCD <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

**Halfword**

```
31 30 29 28 27 26 25 24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  0  0  0  1  0  0  0  1  1  1  imm4  1  1  1  0  0  0  0  pattern  Rdn
```

```
INCH <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```
INCW <Xdn>{, <pattern>{, MUL #<imm>}}</p>

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<p><pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01100</td>
<td>VL64</td>
</tr>
<tr>
<td>01101</td>
<td>VL128</td>
</tr>
<tr>
<td>01110</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uumm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uumm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uumm5</td>
</tr>
<tr>
<td>101101</td>
<td>#uumm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uumm5</td>
</tr>
<tr>
<td>1x0x10</td>
<td>#uumm5</td>
</tr>
<tr>
<td>1xx80</td>
<td>#uumm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(64) operand1 = X[dn];

X[dn] = operand1 + (count * imm);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
INCD, INCH, INCW (vector)

Increment vector by multiple of predicate constraint element count.

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: Doubleword, Halfword and Word

**Doubleword**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 1 1 1 imm4 1 1 0 0 0 0 pattern Zdn
```

INC pitches <Zdn>.D, <pattern>{, MUL #<imm>}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Halfword**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 1 1 1 imm4 1 1 0 0 0 0 pattern Zdn
```

INCH pitches <Zdn>.H, <pattern>{, MUL #<imm>}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Word**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 1 0 1 imm4 1 1 0 0 0 0 pattern Zdn
```

INCW pitches <Zdn>.S, <pattern>{, MUL #<imm>}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10000</td>
<td>MUL4</td>
</tr>
<tr>
<td>10001</td>
<td>MUL4</td>
</tr>
<tr>
<td>10010</td>
<td>MUL4</td>
</tr>
<tr>
<td>10011</td>
<td>MUL4</td>
</tr>
<tr>
<td>10100</td>
<td>MUL4</td>
</tr>
<tr>
<td>10101</td>
<td>MUL4</td>
</tr>
<tr>
<td>10110</td>
<td>MUL4</td>
</tr>
<tr>
<td>10111</td>
<td>MUL4</td>
</tr>
<tr>
<td>11000</td>
<td>MUL4</td>
</tr>
<tr>
<td>11001</td>
<td>MUL4</td>
</tr>
<tr>
<td>11010</td>
<td>MUL4</td>
</tr>
<tr>
<td>11011</td>
<td>MUL4</td>
</tr>
<tr>
<td>11100</td>
<td>MUL4</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL4</td>
</tr>
<tr>
<td>11111</td>
<td>MUL4</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] + (count * imm);
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
INCP (scalar)

Increment scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

INCP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Pm> Is the name of the source scalable predicate register, encoded in the “Pm” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) operand1 = X[dn];
bits(PL) operand2 = P[m];
integer count = 0;

for e = 0 to elements - 1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

X[dn] = operand1 + count;
INCP (vector)

Increment vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

```
INCP <Zdn>.<T>, <Pm>.<T>
```

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] + count;
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
INDEX (immediates)

Create index starting from and incremented by immediate.

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

INDEX <Zd>.<T>, #<imm1>, #<imm2>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer d = Uint(Zd);
integer imm1 = SInt(imm5);
integer imm2 = SInt(imm5b);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm1> Is the first signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

<imm2> Is the second signed immediate operand, in the range -16 to 15, encoded in the “imm5b” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) result;
for e = 0 to elements-1
    integer index = imm1 + e * imm2;
    Elem[result, e, esize] = index<esize-1:0>;

Z[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
INDEX (immediate, scalar)

Create index starting from immediate and incremented by general-purpose register.

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

```
INDEX <Zd>.<T>, #<imm>, <R><m>
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Rm);
integer d = UInt(Zd);
integer imm = SInt(imm5);

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

<R> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(esize) operand2 = X[m];
integer element2 = SInt(operand2);
bits(VL) result;
for e = 0 to elements-1
    integer index = imm + e * element2;
    Elem[result, e, esize] = index<esize-1:0>;
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
INDEX (scalar, immediate)

Create index starting from general-purpose register and incremented by immediate.

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 size 1 imm5 0 1 0 0 0 1 Rn Zd
```

INDEX <Zd>, <T>, <R><n>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
integer imm = SInt(imm5);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the “imm5” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(esize) operand1 = X[n];
integer element1 = SInt(operand1);
bv(VL) result;

for e = 0 to elements-1
    integer index = element1 + e * imm;
    Elem[result, e, esize] = index<esize-1:0>;
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
INDEX (scalars)

Create index starting from and incremented by general-purpose register.

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operands are general-purpose registers in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0   size 1   Rm 0 1 0 0 1 1   Rn   Zd

INDEX <Zd>.<T>, <R><n>, <R><m>

def CheckSVEEnabled():
    integer elements = VL DIV esize;
    bits(esize) operand1 = X[n];
    integer element1 = SInt(operand1);
    bits(esize) operand2 = X[m];
    integer element2 = SInt(operand2);
    bits(VL) result;
    for e = 0 to elements-1
        integer index = element1 + e * element2;
        Elem[result, e, esize] = index<esize-1:0>;
    Z[d] = result;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
INSR (scalar)

Insert general-purpose register in shifted vector.

Shift the destination vector left by one element, and then place a copy of the least-significant bits of the general-purpose register in element 0 of the destination vector. This instruction is unpredicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------------------|-----------------------------|
| 0 0 0 0 0 1 0 1 | size | 1 0 0 1 0 0 0 0 1 1 1 0 | Rm | Zdn |
```

INSR <Zdn>.<T>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Rm);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
bits(VL) dest = Z[dn];
bits(esize) src = X[m];
Z[dn] = dest<VL-esize-1:0> : src;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**INSR (SIMD&FP scalar)**

Insert SIMD&FP scalar register in shifted vector.

Shift the destination vector left by one element, and then place a copy of the SIMD&FP scalar register in element 0 of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>1 1 0 1 0 0 0 1 1 1</th>
<th>Vm</th>
<th>Zdn</th>
</tr>
</thead>
</table>

**INSR** \(<\text{Zdn}>,.<\text{T}>,<\text{V}><\text{m}>\)

if !\text{HaveSVE}() then UNDEFINED;
integer \text{esize} = 8 << \text{UInt}(\text{size});
integer \text{dn} = \text{UInt}(\text{Zdn});
integer \text{m} = \text{UInt}(\text{Vm});

**Assembler Symbols**

\(<\text{Zdn}>\quad \text{Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.}\)

\(<\text{T}>\quad \text{Is the size specifier, encoded in "size":}\)

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;\text{T}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{V}>\quad \text{Is a width specifier, encoded in "size":}\)

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;\text{V}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{m}>\quad \text{Is the number [0-31] of the source SIMD&FP register, encoded in the "Vm" field.}\)

**Operation**

\(\text{CheckSVEEnabled}();\)

\(\text{bits(VL)} \text{ dest} = Z[dn];\)

\(\text{bits(esize)} \text{ src} = V[m];\)

\(Z[dn] = \text{dest<VL-esize-1:0> : src};\)

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LASTA (scalar)

Extract element after last to general-purpose register.

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then zero-extend and place the extracted element in the destination general-purpose register.

```
# LASTA <R><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = if esize < 64 then 32 else 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Rd);
boolean isBefore = FALSE;

Assembler Symbols

<R> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<d> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(rsize) result;
integer last = LastActiveElement(mask, esize);

if isBefore then
  if last < 0 then last = elements - 1;
else
  last = last + 1;
  if last >= elements then last = 0;
result = ZeroExtend(Elem[operand, last, esize]);
X[d] = result;
```
LASTA (SIMD&FP scalar)

Extract element after last to SIMD&FP scalar register.

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then place the extracted element in the destination SIMD&FP scalar register.

```
0 0 0 0 1 1 1 0 0 0 1 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0
```

LASTA \(<V><d>, <Pg>, <Zn>\).<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = FALSE;

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”:

```
size   \(<V>\)
00     B
01     H
10     S
11     D
```

\(<d>\) Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the ”Pg” field.

\(<Zn>\) Is the name of the source scalable vector register, encoded in the ”Zn” field.

\(<T>\) Is the size specifier, encoded in “size”:

```
size   \(<T>\)
00     B
01     H
10     S
11     D
```

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer last = LastActiveElement(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
if last >= elements then last = 0;
V[d] = Elem(operand, last, esize);
```
LASTB (scalar)

Extract last element to general-purpose register.

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then zero-extend and place the extracted element in the destination general-purpose register:

\[
\begin{array}{cccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
B
\end{array}
\]

LASTB \text{<R><d>, <Pg>, <Zn>.<T>}

if !\text{HaveSVE()} then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer rsize = if esize < 64 then 32 else 64;
integer g = \text{UInt}(Pg);
integer n = \text{UInt}(Zn);
integer d = \text{UInt}(Rd);
boolean isBefore = TRUE;

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;R&gt;</th>
<th>Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;R&gt;</td>
</tr>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

| <d> | Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field. |
| <Pg> | Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field. |
| <Zn> | Is the name of the source scalable vector register, encoded in the "Zn" field. |
| <T> | Is the size specifier, encoded in “size”: |
| size | <T> |
| 00   | B   |
| 01   | H   |
| 10   | S   |
| 11   | D   |

Operation

\text{CheckSVEEnabled();}
integer elements = \text{VL DIV esize};
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(rsize) result;
integer last = \text{LastActiveElement}(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
if last >= elements then last = 0;
result = \text{ZeroExtend}(\text{Elem}[operand, last, esize]);
X[d] = result;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LASTB (SIMD&FP scalar)**

Extract last element to SIMD&FP scalar register.

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then place the extracted element in the destination SIMD&FP register.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Pg</th>
<th>Zn</th>
<th>Vd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>1 0 0 0 1 1 1 0 0</td>
<td>0 0 0 0 1 0 0 0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LASTB $<V><d>$, $<Pg>$, $<Zn>$..<T>

if {!HaveSVE()} then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = TRUE;

**Assembler Symbols**

$<V>$ Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;V&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$<d>$ Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

$<Pg>$ Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

$<Zn>$ Is the name of the source scalable vector register, encoded in the "Zn" field.

$<T>$ Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer last = LastActiveElement(mask, esize);

if isBefore then
   if last < 0 then last = elements - 1;
else
   last = last + 1;
if last >= elements then last = 0;
$V[d]$ = Elem(operand, last, esize);
LD1B (vector plus immediate)

Gather load unsigned bytes to vector (immediate index).

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | imm5| 1  | 1  | 0  | Pg | Zn | Zt |

msz<1>msz<0> U ff

LD1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | imm5| 1  | 1  | 0  | Pg | Zn | Zt |

msz<1>msz<0> U ff

LD1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1B (scalar plus immediate)

Contiguous load unsigned bytes to vector (immediate index).

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if ! HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

16-bit element

LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if ! HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if ! HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
64-bit element

LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;

Z[t] = result;
LD1B (scalar plus scalar)

Contiguous load unsigned bytes to vector (scalar index).

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element.

8-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;

16-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

LD1B { <Zt>, <Pg>/Z, [<Xn|SP>, <Xm>] }
LD1B \{ <Zt> . S \}, <Pg>/Z, \[<Xn|SP>, <Xm>] \\
if !HaveSVE() then UNDEFINED;  
if Rm == '11111' then UNDEFINED;  
integer t =UInt(Zt);  
integer n =UInt(Rn);  
integer m =UInt(Rm);  
integer g =UInt(Pg);  
integer esize = 32;  
integer msize = 8;  
boolean unsigned = TRUE;  

### 64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | Rm | 0  | 1  | 0  | Pg | 0  | 1  | 0  | Rn | 1  | Zt |

dtype<3:1>dtype<0>  

LD1B \{ <Zt>.D \}, <Pg>/Z, \[<Xn|SP>, <Xm>] \\
if !HaveSVE() then UNDEFINED;  
if Rm == '11111' then UNDEFINED;  
integer t =UInt(Zt);  
integer n =UInt(Rn);  
integer m =UInt(Rm);  
integer g =UInt(Pg);  
integer esize = 64;  
integer msize = 8;  
boolean unsigned = TRUE;  

### Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.  
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.  
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.  
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

`CheckSVEEnabled();`

```
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
```

if `HaveMTEExt()` then `SetTagCheckedInstruction(TRUE);`

if n == 31 then
  if `LastActiveElement(mask, esize) >= 0 || ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)` then
    `CheckSPAlignment();`
  else
    base = X[n];
else
  base = SP[];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if `ElemP[mask, e, esize] == '1'` then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;

Z[t] = result;
LD1B (scalar plus vector)

Gather load unsigned bytes to vector (vector index).

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset

### 32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | xs | 0  | Zm | 0  | 1  | 0  | Pg | Rn | Zt |    |    |    |    |    |    |    |    |    | U  | ff |

LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 32-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | xs | 0  | Zm | 0  | 1  | 0  | Pg | Rn | Zt |    |    |    |    |    |    |    |    |    | U  | ff |

LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | Zm | 1  | 1  | 0  | Pg | Rn | Zt |    |    |    |    |    |    |    |    |    | U  | ff |

LD1B (scalar plus vector)
LD1B \{ <Zt>.D \}, \langle Pg/Z, [<Xn|SP>, <Zm>.D] \}

if HasSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = int(Elem[off, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1D (vector plus immediate)

Gather load doublewords to vector (immediate index).

Gather load of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff

LD1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];
Elem[result, e, esize] = Extend(data, esize, unsigned);
else
  Elem[result, e, esize] = Zeros();
Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1D (scalar plus immediate)

Contiguous load doublewords to vector (immediate index).

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
  else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = result;
```
**LD1D (scalar plus scalar)**

Contiguous load doublewords to vector (scalar index).

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

![instruction format]

```
LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;

**Assembler Symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAignment();
    base = SP[];
else
    base = X[n];

for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
    offset = offset + 1;

Z[t] = result;
```

LD1D (scalar plus vector)

Gather load doublewords to vector (vector index).

Gather load of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | xs | 1  | 0  | 1  | 0  | Pg | Rn | Zt |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;

32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | xs | 0  | 0  | 1  | 0  | Zm | 0  | 1  | 0  | Pg | Rn | Zt |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | Zm | 1  | 1  | 0  | Pg | Rn | Zt |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

LD1D (scalar plus vector)  Page 1831
LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 1 1 0</td>
</tr>
<tr>
<td>msz&lt;1&gt;msz&lt;0&gt;</td>
</tr>
</tbody>
</table>

LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Rn" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

\[
\text{CheckSVEEnabled();}
\]
\[
\text{integer elements = } \text{VL} \text{ DIV esize;}
\]
\[
\text{bits(64) base;}
\]
\[
\text{bits(64) addr;}
\]
\[
\text{bits(VL) offset = Z[m];}
\]
\[
\text{bits(PL) mask = P[g];}
\]
\[
\text{bits(VL) result;}
\]
\[
\text{bits(msize) data;}
\]
\[
\text{constant integer mbytes = msize DIV 8;}
\]
\[
\text{if } \text{HaveMTEExt()} \text{ then SetTagCheckedInstruction(TRUE);}\]
\[
\text{if } n == 31 \text{ then}
\]
\[
\text{if } \text{LastActiveElement}(\text{mask, esize}) >= 0 || \text{ConstrainUnpredictableBool}(\text{Unpredictable_CHECKSPNONEACTIVE}) \text{ then CheckSPAlignment();}
\]
\[
\text{base = SP[];}
\]
\[
\text{else}
\]
\[
\text{base = X[n];}
\]
\[
\text{for } e = 0 \text{ to elements-1}
\]
\[
\text{if } \text{ElemP(mask, e, esize) == '1' then}
\]
\[
\text{integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);}
\]
\[
\text{addr = base + (off << scale);}
\]
\[
\text{data = Mem[addr, mbytes, AccType_NORMAL];}
\]
\[
\text{Elem[result, e, esize] = Extend(data, esize, unsigned);}\]
\[
\text{else}
\]
\[
\text{Elem[result, e, esize] = Zeros();}
\]
\[
\text{Z[t] = result;}
\]
LD1H (vector plus immediate)

Gather load unsigned halfwords to vector (immediate index).

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  |

msz<1>msz<0> U ff

LD1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### 64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  |

msz<1>msz<0> U ff

LD1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### Assembler Symbols

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the base scalable vector register, encoded in the "Zn" field.
- **<imm>** Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

\textbf{CheckSVEEnabled}();
\begin{verbatim}
integer elements = \texttt{VL} \ 	exttt{DIV} \ 	exttt{esize};
bits(\texttt{VL}) base = \texttt{Z[n]};
bits(64) addr;
bits(PL) mask = \texttt{P[g]};
bits(\texttt{VL}) result;
bits(msize) data;
constant integer mbytes = msize \ 	exttt{DIV} \ 8;
\end{verbatim}

if \texttt{HaveMTEExt}() then \texttt{SetTagCheckedInstruction}(TRUE);

\begin{verbatim}
for e = 0 to elements-1
    if \texttt{ElemP}[mask, e, esize] == '1' then
        addr = \texttt{ZeroExtend}(\texttt{Elem}[base, e, esize], 64) + offset * mbytes;
        data = \texttt{Mem}[addr, mbytes, AccType_NORMAL];
        \texttt{Elem}[result, e, esize] = \texttt{Extend}(data, esize, unsigned);
    else
        \texttt{Elem}[result, e, esize] = \texttt{Zeros}();
\end{verbatim}

\texttt{Z[t]} = result;
LD1H (scalar plus immediate)

Contiguous load unsigned halfwords to vector (immediate index).

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

64-bit element

LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE)
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1H (scalar plus scalar)

Contiguous load unsigned halfwords to vector (scalar index).

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21 20 19 18 17 16 15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 1 0</td>
<td>1</td>
<td>Rm 0 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21 20 19 18 17 16 15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 1 1 0</td>
<td>0</td>
<td>Rm 0 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21 20 19 18 17 16 15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 1 1 0 0 1 0 0</td>
<td>1</td>
<td>Rm 0 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LD1H (scalar plus scalar)
LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;
Z[t] = result;
LD1H (scalar plus vector)

Gather load unsigned halfwords to vector (vector index).

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

```
LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked scaled offset

```
LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked unscaled offset

```
LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
LD1H \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 1 0 0 0 0 1 0 0 1 xs 0                             | Zm 0 1 0                                     | Pg Rn Zt                                     |
| U ff                                               |                                               |                                               |

LD1H \{ <Zt>.S \}, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 1 1 0 0 0 1 0 0 1 1                             | Zm 1 1 0                                     | Pg Rn Zt                                     |
| U ff                                               |                                               |                                               |

LD1H \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;

64-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 1 1 0 0 0 1 0 0 1 0                             | Zm 1 1 0                                     | Pg Rn Zt                                     |
| U ff \text{msz<1>msz<0>}                       |                                               |                                               |
LD1H  { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt>    Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>    Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm>    Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod>   Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        base = SP[];
    else
        base = X[n];
else
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
            addr = base + (off << scale);
            data = Mem[addr, mbytes, AccType_NORMAL];
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        else
            Elem[result, e, esize] = Zeros();
    Z[t] = result;
Load and broadcast unsigned byte to vector.

Load a single unsigned byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element.

### 8-bit element

```
<table>
<thead>
<tr>
<th>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>121110 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm6</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

ldtype<1>ldtype<0> dtypel<1>dtypel<0>
```

LD1RB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{{, #imm}}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
inget msize = 8;
bool unsigned = TRUE;
inget offset = UInt(imm6);

### 16-bit element

```
<table>
<thead>
<tr>
<th>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>121110 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm6</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

ldtype<1>ldtype<0> dtypel<1>dtypel<0>
```

LD1RB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{{, #imm}}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
inget msize = 8;
bool unsigned = TRUE;
inget offset = UInt(imm6);

### 32-bit element

```
<table>
<thead>
<tr>
<th>31302928272625</th>
<th>24</th>
<th>23</th>
<th>2221201918171615</th>
<th>14</th>
<th>13</th>
<th>121110 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>imm6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

ldtype<1>ldtype<0> dtypel<1>dtypel<0>
```

LD1RB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{{, #imm}}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
inget msize = 8;
bool unsigned = TRUE;
inget offset = UInt(imm6);
64-bit element

<table>
<thead>
<tr>
<th>1 0 0 0 0 1 0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>imm6</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtypeh&lt;1&gt;</td>
<td>dtypeh&lt;0&gt;</td>
<td>dtypel&lt;1&gt;</td>
<td>dtypel&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD1RB \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

CheckSVEEnabled();
integer elements = VL \div esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize \div 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
ninteger last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1RD

Load and broadcast doubleword to vector.

Load a single doubleword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 8 in the range 0 to 504.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 504, defaulting to 0, encoded in the "imm6" field.

LD1RD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPA(realignment);
    base = SP[];
else
    base = X[n];
integer last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1RH

Load and broadcast unsigned halfword to vector.

Load a single unsigned halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: **16-bit element**, **32-bit element** and **64-bit element**

**16-bit element**

```plaintext
LD1RH {<Zt>.H}, <Pg>/Z, [<Xn|SP>{, #<imm>}
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

**32-bit element**

```plaintext
LD1RH {<Zt>.S}, <Pg>/Z, [<Xn|SP>{, #<imm>}
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

**64-bit element**

```plaintext
LD1RH {<Zt>.D}, <Pg>/Z, [<Xn|SP>{, #<imm>}
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

Operation

```c
void CheckSVEEnabled()
{
    integer elements = VL DIV esize;
    bits(64) base;
    bits(64) addr;
    bits(PL) mask = P[g];
    bits(VL) result;
    bits(msize) data;
    constant integer mbytes = msize DIV 8;
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    if n == 31 then
        if LastActiveElement(mask, esize) >= 0 ||
            ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
        base = SP[];
    else
        base = X[n];
    integer last = LastActiveElement(mask, esize);
    if last >= 0 then
        addr = base + offset * mbytes;
        data = Mem[addr, mbytes, AccType_NORMAL];
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        else
            Elem[result, e, esize] = Zeros();
    Z[t] = result;
}
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1ROB (scalar plus immediate)

Contiguous load and replicate thirty-two bytes (immediate index).

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

```
ising<1>msz<0> ssz
```

```
LD1ROB {<Zt>.<B>, <Pg>/Z, [<Xn|SP>{, #<imm>}}
```

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
if $VL < 256$ then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = $P[g]$; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if $n == 31$ then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = $SP[]$;
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = $X[n]$;

addr = base + offset * 32;
for $e = 0$ to elements-1
  if $ElemP[mask, e, esize] == '1'$ then
    $Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];$
  else
    $Elem[result, e, esize] = Zeros();$
  addr = addr + mbytes;

$Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);$
**LD1ROB (scalar plus scalar)**

Contiguous load and replicate thirty-two bytes (scalar index).

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

```plaintext
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
msz<1>msz<0> ssz
```

LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];

offset = X[m];

addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
**LD1ROD (scalar plus immediate)**

Contiguous load and replicate four doublewords (immediate index).

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

```plaintext
LD1ROD {<Zt>}.D, <Pg>/Z, [<Xn|SP>{, #<imm>}
```

if !HaveSVEFP64MatMulExt() then UNDEFINED;

integer \( t = \text{UInt}(Zt); \)

double \( n = \text{UInt}(Rn); \)

double \( g = \text{UInt}(Pg); \)

double \( \text{esize} = 64; \)

integer \( \text{offset} = \text{SInt}(\text{imm4}); \)

**Assembler Symbols**

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

LD1ROD (scalar plus immediate)
Operation

CheckSVEEnabled();
if \( VL < 256 \) then UNDEFINED;
integer elements = 256 DIV esize;
bhits(64) base;
bhits(64) addr;
bhits(PL) mask = \( P[g] \); // low bits only
bhits(256) result;
constant integer mbytes = esize DIV 8;

if \( n == 31 \) then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = \( SP[] \);
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = \( X[n] \);

addr = base + offset * 32;
for e = 0 to elements-1
  if \( \text{Elem}[\text{mask}, e, \text{esize}] == '1' \) then
    \( \text{Elem}[	ext{result}, e, \text{esize}] = \text{Mem}[	ext{addr}, \text{mbytes}, \text{AccType}\_\text{NORMAL}] \);
  else
    \( \text{Elem}[	ext{result}, e, \text{esize}] = \text{Zeros}() \);
  addr = addr + mbytes;

\( Z[t] = \text{ZeroExtend(Replicate(result, \( VL \)\ DIV 256), \( VL \})} \);
**LD1ROD (scalar plus scalar)**

Contiguous load and replicate four doublewords (scalar index).

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

```assembly
LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
```

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

`CheckSVEEnabled();`
if $VL < 256$ then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(64) addr;
bits($PL$) mask = $P[g]$; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if $\text{HaveMTEExt}()$ then $\text{SetTagCheckedInstruction}(\text{TRUE})$;

if $n == 31$ then
  if $\text{LastActiveElement}(mask, esize) >= 0$ ||
     $\text{ConstrainUnpredictableBool}(\text{Unpredictable}_\text{CHECKSPNONEACTIVE})$ then
    $\text{CheckSPAlignment}();$
  else
    base = $SP[]$;
  else
    base = $X[n]$;

offset = $X[m]$;

addr = base + UInt(offset) * mbytes;
for $e = 0$ to elements-1
  if $\text{ElemP}[mask, e, esize] == '1'$ then
    $\text{Elem}[result, e, esize] = \text{Mem}[addr, mbytes, AccType\_NORMAL]$;
  else
    $\text{Elem}[result, e, esize] = \text{Zeros}();$
  addr = addr + mbytes;

$Z[t] = \text{ZeroExtend}(\text{Replicate}(result, VL\_DIV\_256), VL)$;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09.rc2b; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1ROH (scalar plus immediate)

Contiguous load and replicate sixteen halfwords (immediate index).

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

```
LD1ROH {<Zt>.H},<Pg>/Z, [<Xn|SP>{, #<imm>}
```

if !HaveSVEFP64MatMulExt() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);

Assembler Symbols

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
if \(VL < 256\) then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * 32;
for e = 0 to elements - 1
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
LD1ROH (scalar plus scalar)

Contiguous load and replicate sixteen halfwords (scalar index).

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

![Assembler Symbols]

LD1ROH (scalar plus scalar)

LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1] if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  base = SP[];
else
  base = X[n];

offset = X[m];
addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
LD1ROW (scalar plus immediate)

Contiguous load and replicate eight words (immediate index).

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

```assembly
LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
```
Operation

```c
CheckSVEEnabled();
if VL < 256 then UNDEFINED;
integer elements = 256 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * 32;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;

Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1ROW (scalar plus scalar)

Contiguous load and replicate eight words (scalar index).

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero. Only the first eight predicate elements are used and higher numbered predicate elements are ignored. ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 | 1 | 0 | 0 | 1 | Rm | 0 | 0 | 0 | Pg | Rn | Zt
msz<1>msz<0> ssz
```

LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = Uint(Zt);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer g = Uint(Pg);
integer esize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
if \( VL < 256 \) then UNDEFINED;
integer elements = 256 DIV \( \text{esize} \);
bits(64) base;
bits(64) addr;
bits(PL) mask = \( P[g] \); // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = \( \text{esize} \) DIV 8;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if \( n == 31 \) then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = \( SP[] \);
else
  base = \( X[n] \);

offset = \( X[m] \);

addr = base + UInt(offset) * mbytes;
for \( e = 0 \) to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = \( \text{Mem}[\text{addr, mbytes, AccType_NORMAL}] \);
  else
    Elem[result, e, esize] = \( Zeros() \);
  addr = addr + mbytes;

\( Z[t] = \text{ZeroExtend(Replicate(result, VL DIV 256), VL)} \);
LD1RQB (scalar plus immediate)

Contiguous load and replicate sixteen bytes (immediate index).

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

![Binary representation](image)

LD1RQB \{ <Zt>.B \}, <Pg>/Z, [<Xn|SP>{, #<imm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

Assembler Symbols

- **<Zt>** is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<imm>** is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * 16;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;

Z[t] = Replicate(result, VL DIV 128);
LD1RQB (scalar plus immediate)
**LD1RQB (scalar plus scalar)**

Contiguous load and replicate sixteen bytes (scalar index).

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Rm Pg Rn Zt
```

```
LD1RQB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

**Assembler Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    base = SP[];
  else
    base = X[n];
  offset = X[m];
addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
```

LD1RQB (scalar plus scalar)
LD1RQD (scalar plus immediate)

Contiguous load and replicate two doublewords (immediate index).

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 1 1 0 0 0 0 1 0 0 1 0 1 0 0 1 0 1 1 0 0 0 0 imm4 0 0 1 Pg Rn Zt

LD1RQD {<Zt>..D}, <Pg>/Z, [<Xn|SP>{, #<imm>}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * 16;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
LD1RQD (scalar plus scalar)

Contiguous load and replicate two doublewords (scalar index).

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | Rm | 0 | 0 | 0 | Pg | Rn | Zt |

msz<1> msz<0> ssz

LD1RQD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstranUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = X[m];

addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;

Z[t] = Replicate(result, VL DIV 128);
**LD1RQH (scalar plus immediate)**

Contiguous load and replicate eight halfwords (immediate index).

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

```
LD1RQH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);

**Assembler Symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
  else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
addr = base + offset * 16;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
```
LD1RQH (scalar plus scalar)

Contiguous load and replicate eight halfwords (scalar index).

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

```
LD1RQH { <Zt>.H }, <Pg>/Z, [ <Xn|SP> ], <Xm>, LSL #1
```

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[ ];
else
    base = X[n];
offset = X[m];
addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
```
**LD1RQW (scalar plus immediate)**

Contiguous load and replicate four words (immediate index).

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

![Assembler Symbols](image)

**Operation**

```
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);

if !HaveSVE() then UNDEFINED;
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * 16;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = Replicate(result, VL DIV 128);
```
LD1RQW (scalar plus scalar)

Contiguous load and replicate four words (scalar index).

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalable base address and scalar index which is multiplied by 4 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------------------------------------------------|------------------|
| 1 0 1 0 0 1 0 | 1 0 0 0       | Rm | 0 0 0 | Pg | Rn | Zt |

LD1RQW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = 128 DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
offset = X[m];
addr = base + UInt(offset) * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;

Z[t] = Replicate(result, VL DIV 128);
LD1RQW (scalar plus scalar)
LD1RSB

Load and broadcast signed byte to vector.

Load a single signed byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element.

16-bit element

![Binary representation of LD1RSB instruction in 16-bit element]

LD1RSB \{ <Zt> . H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]  
if !HaveSVE() then UNDEFINED;  
integer t = UInt(Zt);  
integer n = UInt(Rn);  
integer g = UInt(Pg);  
integer esize = 16;  
integer msize = 8;  
boolean unsigned = FALSE;  
integer offset = UInt(imm6);

32-bit element

![Binary representation of LD1RSB instruction in 32-bit element]

LD1RSB \{ <Zt> . S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]  
if !HaveSVE() then UNDEFINED;  
integer t = UInt(Zt);  
integer n = UInt(Rn);  
integer g = UInt(Pg);  
integer esize = 32;  
integer msize = 8;  
boolean unsigned = FALSE;  
integer offset = UInt(imm6);

64-bit element

![Binary representation of LD1RSB instruction in 64-bit element]

LD1RSB \{ <Zt> . D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]  
if !HaveSVE() then UNDEFINED;  
integer t = UInt(Zt);  
integer n = UInt(Rn);  
integer g = UInt(Pg);  
integer esize = 64;  
integer msize = 8;  
boolean unsigned = FALSE;  
integer offset = UInt(imm6);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PPL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 || ConstrainsUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAAlignment();
    base = SP[];
else
    base = X[n];
integer last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD1RSH**

Load and broadcast signed halfword to vector:

Load a single signed halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 0 1 1 0 | imm6 | 1 0 1 | Pg | Rn | Zt
```

```
LD1RSH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{{, #<imm>}}]
```

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

### 64-bit element

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 0 1 1 0 | imm6 | 1 0 0 | Pg | Rn | Zt
```

```
LD1RSH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{{, #<imm>}}]
```

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

integer last = LastActiveElement(mask, esize);
if last >= 0 then
  addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1RSW

Load and broadcast signed word to vector.

Load a single signed word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        base = SP[];
else
    base = X[n];

integer last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1RW

Load and broadcast unsigned word to vector.

Load a single unsigned word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.

Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 10000101 | 0 | 1 | 0 | imm6 | 1 | 1 | 0 | Pg | Rn | Zt |

```
LD1RW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}
```

if !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

### 64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 10000101 | 1 | 0 | 1 | imm6 | 1 | 1 | 1 | Pg | Rn | Zt |

```
LD1RW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}
```

if !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

### Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.
Operation

`CheckSVEEnabled();`
integer elements = $VL$ DIV esize;
bits(64) base;
bits(64) addr;
bits($PL$) mask = $P[g]$;
bits($VL$) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    else
        base = SP[];
else
    base = $X[n]$;

integer last = LastActiveElement(mask, esize);
if last >= 0 then
    addr = base + offset * mbytes;
data = Mem[addr, mbytes, AccType_NORMAL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();

$Z[t] = result;$
LD1SB (vector plus immediate)

Gather load signed bytes to vector (immediate index).

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

```plaintext
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |
```

LD1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### 64-bit element

```plaintext
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
```

LD1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### Assembler Symbols

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
LD1SB (scalar plus immediate)

Contiguous load signed bytes to vector (immediate index).

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 1 0 0</td>
</tr>
</tbody>
</table>

LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t =UInt(Zt);
integer n =UInt(Rn);
integer g =UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 0 1 0</td>
</tr>
</tbody>
</table>

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t =UInt(Zt);
integer n =UInt(Rn);
integer g =UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 0 0 0</td>
</tr>
</tbody>
</table>

LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t =UInt(Zt);
integer n =UInt(Rn);
integer g =UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

\[
\text{CheckSVEEnabled();}
\]

integer elements = VL \text{ DIV} \text{ esize};

bits(64) base;

bits(64) addr;

bits(PL) mask = P[g];

bits(VL) result;

bits(msize) data;

constant integer mbytes = msize \text{ DIV} \text{ 8};

if \ n == 31 then
  if \ \text{LastActiveElement}(mask, \text{ esize}) >= 0 ||
    \text{ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE)} then
    \text{CheckSPAlignment();}
  \text{if \ HaveMTEExt() then SetTagCheckedInstruction(FALSE);}
  base = SP[];
else
  if \ \text{HaveMTEExt() then SetTagCheckedInstruction(TRUE);}
  base = X[n];

addr = base + offset * elements * mbytes;

for e = 0 to elements - 1
  if \ \text{ElemP}(mask, e, \text{ esize}) == '1' then
    data = \text{Mem}(addr, mbytes, \text{ AccType\_NORMAL});
    \text{Elem}(result, e, \text{ esize}) = \text{Extend}(data, \text{ esize}, \text{ unsigned});
  else
    \text{Elem}(result, e, \text{ esize}) = \text{Zeros}();
    addr = addr + mbytes;

Z[t] = result;
**LD1SB (scalar plus scalar)**

Contiguous load signed bytes to vector (scalar index).

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: **16-bit element**, **32-bit element** and **64-bit element**

### 16-bit element

```
1 0 1 0 0 1 0 1 1 0 0 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 0
```

**dtype<3:1> dtype<0>**

```
LD1SB {<Zt>.H}, <Pg>/Z, [<Xn|SP>, <Xm>]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;

### 32-bit element

```
1 0 1 0 0 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0
```

**dtype<3:1> dtype<0>**

```
LD1SB {<Zt>.S}, <Pg>/Z, [<Xn|SP>, <Xm>]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;

### 64-bit element

```
1 0 1 0 0 1 0 1 1 0 0 1 0 1 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0
```

**dtype<3:1> dtype<0>**
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;

Z[t] = result;
LD1SB (scalar plus vector)

Gather load signed bytes to vector (vector index).

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset.

32-bit unpacked unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 1 0 0 0 1 0 0 0 xs 0 Zm 0 0 0 Pg Rn Zt |
| msz<1>msz<0> U ff |

LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == ‘0’;
integer scale = 0;

32-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 0 0 0 0 1 0 0 0 xs 0 Zm 0 0 0 Pg Rn Zt |
| U ff |

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == ‘0’;
integer scale = 0;

64-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 1 0 0 0 1 0 0 0 1 0 Zm 1 0 0 Pg Rn Zt |
| msz<1>msz<0> U ff |

LD1SB (scalar plus vector)
LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm>  Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod>  Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) == 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        integer off = int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
**LD1SH (vector plus immediate)**

Gather load signed halfwords to vector (immediate index).

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | imm5 | 1  | 0  | 0  | Pg | Zn | Zt |

LD1SH { <Zt>.S }, <Pg>/Z, {<Zn>.S{, #<imm>}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### 64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | imm5 | 1  | 0  | 0  | Pg | Zn | Zt |

LD1SH { <Zt>.D }, <Pg>/Z, {<Zn>.D{, #<imm>}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

\begin{verbatim}
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;
\end{verbatim}
LD1SH (scalar plus immediate)

Contiguous load signed halfwords to vector (immediate index).

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInT(Zt);
integer n = UInT(Rn);
integer g = UInT(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInT(imm4);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInT(Zt);
integer n = UInT(Rn);
integer g = UInT(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInT(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

\textbf{CheckSVEEnabled}();
integer elements = \textbf{VL} \text{ DIV } \text{esize};
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize \text{ DIV } 8;

if n == 31 then
  if \textbf{LastActiveElement}(mask, esize) \geq 0 ||
     \textbf{ConstrainUnpredictableBool}(\text{Unpredictable_CHECKSPNONEACTIVE}) \text{ then }
    \textbf{CheckSPAlignment}();
  if \textbf{HaveMTEExt}() \text{ then } \textbf{SetTagCheckedInstruction}(FALSE);
  base = SP[];
else
  if \textbf{HaveMTEExt}() \text{ then } \textbf{SetTagCheckedInstruction}(TRUE);
  base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if \textbf{ElemP}(mask, e, esize) \text{ == '1' then }
    data = \textbf{Mem}[addr, mbytes, \text{AccType_NORMAL}];
    \textbf{Elem}[result, e, esize] = \textbf{Extend}(data, esize, unsigned);
  else
    \textbf{Elem}[result, e, esize] = \textbf{Zeros}();
  addr = addr + mbytes;

Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1SH (scalar plus scalar)

Contiguous load signed halfwords to vector (scalar index).

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAAlignment();
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + Uint(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;

Z[t] = result;
```
**LD1SH (scalar plus vector)**

Gather load signed halfwords to vector (vector index).

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

### 32-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | xs | 1 |
| U | ff |

**LD1SH** { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

if !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

### 32-bit unpacked scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | xs | 1 |
| U | ff |

**LD1SH** { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

if !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

### 32-bit unpacked unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | xs | 0 |
| msz<1>msz<0> | U | ff |
LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 0 0 0 1 0 0 1 xs 0              | Zm 0 0 0                           | Pg Rn Zt                         |

LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 0 0 0 1 0 1 1 1              | Zm 1 0 0                           | Pg Rn Zt                         |

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;

64-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 0 0 0 1 0 1 1               | Zm 1 0 0                           | Pg Rn Zt                         |

LD1SH (scalar plus vector)
LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
LD1SW (vector plus immediate)

Gather load signed words to vector (immediate index).

Gather load of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

LD1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1SW (scalar plus immediate)

Contiguous load signed words to vector (immediate index).

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 0 1 0 0 1 0 0 0 0 imm4 1 0 1 Pg Rn Zt

dtype<3:1>dtype<0>
```

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if Elem[mask, e, esize] == '1' then
        data = Mem[addr, mbytes, AccType_NORMAL];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;
Z[t] = result;
```
**LD1SW (scalar plus scalar)**

Contiguous load signed words to vector (scalar index).

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>dtype&lt;3:1&gt;dtype&lt;0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;

if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;

Z[t] = result;
LD1SW (scalar plus vector)

Gather load signed words to vector (vector index).

Gather load of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: **32-bit unpacked scaled offset**, **32-bit unpacked unscaled offset**, **64-bit scaled offset** and **64-bit unscaled offset**

### 32-bit unpacked scaled offset

```
 0 0 0 1 0 1 0 | xs | 1 | Zm  | 0 0 0 | Pg  | Rn  | Zt  
```

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

### 32-bit unpacked unscaled offset

```
 0 0 0 1 0 1 0 | xs | 0 | Zm  | 0 0 0 | Pg  | Rn  | Zt  
```

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit scaled offset

```
 0 0 0 1 0 1 0 | 1 | Zm  | 1 0 0 | Pg  | Rn  | Zt  
```

LD1SW (scalar plus vector)
LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|---------------|
| 1 1 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 |
| Zm | Pg | Rn | Zt |
| msz<1>msz<0> | U | f |

LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Rn" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(64) base;
bits(64) addr;
bits(\texttt{VL}) offset = \texttt{Z}[m];
bits(PL) mask = \texttt{P}[g];
bits(\texttt{VL}) result;
bits(msize) data;
constant integer mbytes = msize \texttt{DIV} 8;

if \texttt{HaveMTEExt}() then \texttt{SetTagCheckedInstruction}(TRUE);

if n == 31 then
  if \texttt{LastActiveElement}(mask, esize) >= 0 ||
    \texttt{ConstrainUnpredictableBool}(\texttt{Unpredictable\_CHECKSPNONEACTIVE}) then
    \texttt{CheckSPA}\texttt{Alignment}();
  base = \texttt{SP}[];
else
  base = \texttt{X}[n];

for e = 0 to elements-1
  if \texttt{ElemP}[mask, e, esize] == '1' then
    integer off = \texttt{Int}(\texttt{Elem}[offset, e, esize]<offs\_size-1:0>, offs\_unsigned);
    addr = base + (off << scale);
    data = \texttt{Mem}[addr, mbytes, AccType\_NORMAL];
    \texttt{Elem}[result, e, esize] = \texttt{Extend}(data, esize, unsigned);
  else
    \texttt{Elem}[result, e, esize] = \texttt{Zeros}();

\texttt{Z}[t] = result;
LD1W (vector plus immediate)

Gather load unsigned words to vector (immediate index).

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 1 0 0 1</td>
</tr>
</tbody>
</table>

LD1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### 64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 0 0 1</td>
</tr>
</tbody>
</table>

LD1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

### Assembler Symbols

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.
**Operation**

```plaintext
CheckSVEenabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD1W (scalar plus immediate)**

Contiguous load unsigned words to vector (immediate index).

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

```
0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
imm4 Pg Rn Zt
```

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### 64-bit element

```
0 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
imm4 Pg Rn Zt
```

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

### Assembler Symbols

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
Z[t] = result;
**LD1W (scalar plus scalar)**

Contiguous load unsigned words to vector (scalar index).

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 1 0 0 | 1 0 1 0 | 0 0 1 0 1 0 0 1 0 0 1 0 | Rm | 0 1 0 | Pg | Rn | Zt |

dtype<3:1> dtype<0>

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;

### 64-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 1 0 0 | 1 0 1 0 | 1 0 1 0 | Rm | 1 0 1 0 | Pg | Rn | Zt |

dtype<3:1> dtype<0>

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;

### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

\textbf{CheckSVEEnabled();}
integer elements = \textit{VL} \text{ DIV} esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = \textit{P}[g];
bits(\textit{VL}) result;
bits(msize) data;
bits(64) offset = \textit{X}[m];
constant integer mbytes = msize \text{ DIV} 8;

if \textit{HaveMTEExt()} then \textit{SetTagCheckedInstruction}(TRUE);

if n == 31 then
  if \textit{LastActiveElement}(mask, esize) >= 0 ||
      \textit{ConstrainUnpredictableBool}(Unpredictable\_CHECKSPNONEACTIVE) then
    \textit{CheckSPAignment}();
  base = \textit{SP}[];
else
  base = \textit{X}[n];

for e = 0 to elements-1
  addr = base + \textit{UInt}(offset) * mbytes;
  if \textit{Elem}[mask, e, esize] == '1' then
    data = \textit{Mem}[addr, mbytes, \textit{AccType\_NORMAL}];
    \textit{Elem}[result, e, esize] = \textit{Extend}(data, esize, unsigned);
  else
    \textit{Elem}[result, e, esize] = \textit{Zeros}();
  offset = offset + 1;
\textit{Z}[t] = result;
LD1W (scalar plus vector)

Gather load unsigned words to vector (vector index).

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

32-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 1 0 0 0 0 1 0 1 0 xs| 1 | Zm | 0 | 1 | 0 | Pg | Rn | Zt |

32-bit unpacked scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 1 1 0 0 0 1 0 1 0 xs| 1 | Zm | 0 | 1 | 0 | Pg | Rn | Zt |

32-bit unpacked unscaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 1 1 0 0 0 1 0 1 0 xs| 0 | Zm | 0 | 1 | 0 | Pg | Rn | Zt |

U ff

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

LD1W (scalar plus vector)

Page 1920
LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 32-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | | | | | | | | | | | | | | | | | | | | | |

LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | | | | | | | | | | | | | | | | | | | | | | |

LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;

### 64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | | | | | | | | | | | | | | | | | | | | | |

LD1W (scalar plus vector)
LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
    base = SP[];
  else
    base = X[n];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    integer off = Int(Elem[<offset, e, esize><offs_size-1:0>, offs_unsigned]);
    addr = base + (off << scale);
    data = Mem[addr, mbytes, AccType_NORMAL];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;

Internal version only: isa v32.12,AdvSIMD v29.04,pseudocode v2020-09_xml,sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD2B (scalar plus immediate)

Contiguous load two-byte structures to two vectors (immediate index).

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\[\begin{array}{ccccccccccccccccccccccc}
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \text{imm4} & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt}
\end{array}\]

LD2B \{<Zt1>.B, <Zt2>.B \}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PU) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
        else
            Elem[values[r], e, esize] = Zeros();
        addr = addr + mbytes;
for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LD2B (scalar plus scalar)**

Contiguous load two-byte structures to two vectors (scalar index).

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

![Binary representation of LD2B instruction](image)

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

```c
LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [ <Xn|SP>, <Xm> ]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 2;
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAignment();
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD2D (scalar plus immediate)

Contiguous load two-doubleword structures to two vectors (immediate index).

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

```
LD2D { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 2;

**Assembler Symbols**

- **<Zt1>** Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- **<Zt2>** Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<imm>** Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
     ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD2D (scalar plus scalar)

Contiguous load two-doubleword structures to two vectors (scalar index).

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 0 1 0 0 1 0 | 1 | 1 | 0 | 1 | Rm | 1 1 0 | Pg | Rn | Zt |
```

LD2D {<Zt1>.D, <Zt2>.D}, <Pg> / Z, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrantUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD2H (scalar plus immediate)

Contiguous load two-halfword structures to two vectors (immediate index).

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

Assembler Symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm>  Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```
LD2H (scalar plus scalar)

Contiguous load two-halfword structures to two vectors (scalar index).

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

LD2H { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 2;

Assembler Symbols

- **<Zt1>** is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- **<Zt2>** is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- **<Pg>** is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<Xm>** is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = X[n];
else
  base = SP[];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
**LD2W (scalar plus immediate)**

Contiguous load two-word structures to two vectors (immediate index).

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

![Binary Representation](image)

LD2W \{ `<Zt1>.S`, `<Zt2>.S` \}, `<Pg>/Z`, \[ `<Xn|SP>`{, `<imm>`, MUL VL}\}

if !HaveSVE() then UNDEFINED;

integer t =.UInt(Zt);
integer n =.UInt(Rn);
integer g =.UInt(Pg);
integer esize = 32;
integer offset =.SInt(imm4);
integer nreg = 2;

**Assembler Symbols**

- `<Zt1>` is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
  for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD2W (scalar plus scalar)

Contiguous load two-word structures to two vectors (scalar index).

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.

```
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 2;
```

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the “Zt” field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
     ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;
  
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
**LD3B (scalar plus immediate)**

Contiguous load three-byte structures to three vectors (immediate index).

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

![Assembly Code](image)


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
integer nreg = 3;

**Assembler Symbols**

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = \texttt{P}[g];
constant integer mbytes = esize \texttt{DIV} 8;
array [0..2] of bits(\texttt{VL}) values;

if n == 31 then
    if \texttt{LastActiveElement}(mask, esize) >= 0 ||
        \texttt{ConstrainUnpredictableBool}(Unpredictable\_CHECKSPNONEACTIVE) then
        \texttt{CheckSPAlignment}();
    if \texttt{HaveMTEExt}() then \texttt{SetTagCheckedInstruction}(FALSE);
    base = \texttt{SP}[];
else
    if \texttt{HaveMTEExt}() then \texttt{SetTagCheckedInstruction}(TRUE);
    base = \texttt{X}[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if \texttt{ElemP}[mask, e, esize] == '1' then
            \texttt{Elem}[values[r], e, esize] = \texttt{Mem}[addr, mbytes, \texttt{AccType\_NORMAL}];
        else
            \texttt{Elem}[values[r], e, esize] = \texttt{Zeros}();
        addr = addr + mbytes;
for r = 0 to nreg-1
    \texttt{Z}[(t+r) \texttt{MOD} 32] = values[r];
**LD3B (scalar plus scalar)**

Contiguous load three-byte structures to three vectors (scalar index).

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```plaintext
LD3B (scalar plus scalar)

Contiguous load three-byte structures to three vectors (scalar index).

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.
```

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
   if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   base = SP[];
else
   base = X[n];

for e = 0 to elements-1
   addr = base + UInt(offset) * mbytes;
   for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1' then
         Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
      else
         Elem[values[r], e, esize] = Zeros();
      addr = addr + mbytes;
      offset = offset + nreg;

for r = 0 to nreg-1
   Z[(t+r) MOD 32] = values[r];
**LD3D (scalar plus immediate)**

Contiguous load three-doubleword structures to three vectors (immediate index).

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 1 1 1 0 0 imm4 1 1 1 Pg Rn Zt
msz<1>msz<0>
```

LD3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 3;
```

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD3D (scalar plus scalar)

Contiguous load three-doubleword structures to three vectors (scalar index).

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

\[
\begin{array}{cccccccccccccccccccccccc}
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & Rm & 1 & 1 & 0 & Pg & Rn & Zt \\
msz<1> & msz<0>
\end{array}
\]

LD3D \{ <Zt1>.D, <Zt2>.D, <Zt3>.D \}, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer nreg = 3;

Assembler Symbols

- <Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- <Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- <Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- <Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();

  addr = addr + mbytes;
  offset = offset + nreg;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
**LD3H (Scalar plus Immediate)**

Contiguous load three-halfword structures to three vectors (immediate index).

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
1 0 1 0 0 1 0 | 0 | 1 | 1 0 0 | imm4 | 1 1 1 | Pg | Rn | Zt
```

msz<1>msz<0>


```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 3;
```

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
  for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
**LD3H (scalar plus scalar)**

Contiguous load three-halfword structures to three vectors (scalar index).

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

The instruction is:

```
LD3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]
```

If `!HaveSVE()` then UNDEFINED;
if `Rm == '11111'` then UNDEFINED;
integer `t = UInt(Zt)`;
integer `n = UInt(Rn)`;
integer `m = UInt(Rm)`;
integer `g = UInt(Pg)`;
integer `esize = 16`
integer `nreg = 3`;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrantUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

**LD3W (scalar plus immediate)**

Contiguous load three-word structures to three vectors (immediate index).

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | imm4 | 1  | 1  | 1  | Pg  | Rn  | Zt  |

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

```plaintext
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 3;
```
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
        else
            Elem[values[r], e, esize] = Zeros();
        addr = addr + mbytes;
for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD3W (scalar plus scalar)

Contiguous load three-word structures to three vectors (scalar index).

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

LD3W {<Zt1>.S, <Zt2>.S, <Zt3>.S}, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 3;

msz<1>msz<0>

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 || ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
      addr = addr + mbytes;
      offset = offset + nreg;
    for r = 0 to nreg-1
      Z[(t+r) MOD 32] = values[r];
```
LD4B (scalar plus immediate)

Contiguous load four-byte structures to four vectors (immediate index).

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

LD4B (scalar plus immediate)

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>  Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>  Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Elem(values[r], e, esize) = Mem[addr, mbytes, AccType_NORMAL];
        else
            Elem(values[r], e, esize) = Zeros();
        addr = addr + mbytes;
    for r = 0 to nreg-1
        Z[(t+r) MOD 32] = values[r];
```
LD4B (scalar plus scalar)

Contiguous load four-byte structures to four vectors (scalar index).

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 4;
```

**Assembler Symbols**

- `<Zt1>`: Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>`: Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>`: Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>`: Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];

for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
        else
            Elem[values[r], e, esize] = Zeros();
        addr = addr + mbytes;
        offset = offset + nreg;

for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```
LD4D (scalar plus immediate)

Contiguous load four-doubleword structures to four vectors (immediate index).

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

```
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbuckets = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainingUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;

for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD4D (scalar plus scalar)

Contiguous load four-doubleword structures to four vectors (scalar index).

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.

<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  base = SP[];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
    addr = addr + mbytes;
    offset = offset + nreg;
for r = 0 to nreg-1
  Z[(t+r) MOD 32] = values[r];
LD4H (scalar plus immediate)

Contiguous load four-halfword structures to four vectors (immediate index).

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
    else
      Elem[values[r], e, esize] = Zeros();
      addr = addr + mbytes;
  for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD4H (scalar plus scalar)

Contiguous load four-halfword structures to four vectors (scalar index).

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

LD4H (scalar plus scalar) Page 1965
Operation

```c
CheckSVEEnabled();
integer elements = \texttt{VL} \div \texttt{esize};
bits(64) base;
bits(64) addr;
bits(PL) mask = \texttt{P}[g];
bits(64) offset = \texttt{X}[m];
constant integer mbytes = \texttt{esize} \div 8;
array [0..3] of bits(\texttt{VL}) values;
if \texttt{HaveMTEExt()} then \texttt{SetTagCheckedInstruction}(\text{TRUE});
if n == 31 then
  if \texttt{LastActiveElement}(mask, esize) >= 0 ||
      \texttt{ConstrainUnpredictableBool}(\texttt{Unpredictable_CHECKSPNONEACTIVE}) then
    \texttt{CheckSPAIDgment}();
  else
    base = \texttt{SP}[];
else
  base = \texttt{X}[n];
for e = 0 to elements - 1
  addr = base + \texttt{UInt}(offset) \times \texttt{mbytes};
  for r = 0 to nreg - 1
    if \texttt{ElemP}[mask, e, esize] == '1' then
      \texttt{Elem}[values[r], e, esize] = \texttt{Mem}[addr, mbytes, \texttt{AccType_NORMAL}];
    else
      \texttt{Elem}[values[r], e, esize] = \texttt{Zeros}();
      addr = addr + mbytes;
      offset = offset + nreg;
  for r = 0 to nreg - 1
    \texttt{Z}[(t+r) \mod 32] = values[r];
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD4W (scalar plus immediate)

Contiguous load four-word structures to four vectors (immediate index).

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication. Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 1 | 0 1 1 0 | imm4 | 1 1 1 | Pg | Rn | Zt


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize \texttt{DIV} 8;
array [0..3] of bits(\texttt{VL}) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable\_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = \texttt{SP}[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if \texttt{ElemP[mask, e, esize]} == '1' then
      \texttt{Elem}[values[r], e, esize] = \texttt{Mem}[addr, mbytes, AccType\_NORMAL];
    else
      \texttt{Elem}[values[r], e, esize] = \texttt{Zeros}();
    addr = addr + mbytes;
  for r = 0 to nreg-1
    \texttt{Z}[(t+r) MOD 32] = values[r];
LD4W (scalar plus scalar)

Contiguous load four-word structures to four vectors (scalar index).

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.


if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 4;
Operation

```c
CheckSVEEnabled();
integer elements = VL \div esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize \div 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) == 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];

for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_NORMAL];
        else
            Elem[values[r], e, esize] = Zeros();
        addr = addr + mbytes;
        offset = offset + nreg;

for r = 0 to nreg-1
    Z[(t+r) MOD 32] = values[r];
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDFF1B (vector plus immediate)

Gather load first-fault unsigned bytes to vector (immediate index).

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | imm5| 1  | 1  | 1  | Pg | Zn | Zt | msz<1> | msz<0> | U ff |

LDFF1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | imm5| 1  | 1  | 1  | Pg | Zn | Zt | msz<1> | msz<0> | U ff |

LDFF1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \text{ DIV} \texttt{esize};
bits(\texttt{VL}) base = \texttt{Z}[n];
bits(64) addr;
bits(\texttt{PL}) mask = \texttt{P}[g];
bits(\texttt{VL}) result;
bits(\texttt{VL}) orig = \texttt{Z}[t];
bits(msize) data;
constant integer mbytes = msize \text{ DIV} 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if \texttt{HaveMTEExt}() then \texttt{SetTagCheckedInstruction}(TRUE);
for e = 0 to elements-1
  if \texttt{ElemP[mask, e, esize]} == '1' then
    addr = \texttt{ZeroExtend(Elem[base, e, esize], 64)} + offset \times mbytes;
    if first then
      \texttt{// Mem[]} will not return if a fault is detected for the first active element}
      \texttt{data} = \texttt{Mem[addr, mbytes, AccType\_NORMAL]};
      \texttt{first} = FALSE;
    else
      \texttt{// MemNF[]} will return fault=TRUE if access is not performed for any reason}
      \texttt{(data, fault)} = \texttt{MemNF[addr, mbytes, AccType\_NONFAULT]};
  else
    \texttt{(data, fault)} = \texttt{(Zeros(msize), FALSE)};
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted \mid \text{ fault};
  if faulted then
    \texttt{ElemFFR[e, esize]} = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown \mid \texttt{ElemFFR[e, esize]} == '0';
  if unknown then
    if !fault \&\& \texttt{ConstrainUnpredictableBool(\texttt{Unpredictable\_SVELDNFDATA})} then
      \texttt{Elem[result, e, esize]} = \texttt{Extend(data, esize, unsigned)};
    elseif \texttt{ConstrainUnpredictableBool(\texttt{Unpredictable\_SVELDNFZERO})} then
      \texttt{Elem[result, e, esize]} = \texttt{Zeros()};
    else // merge
      \texttt{Elem[result, e, esize]} = \texttt{Elem[orig, e, esize]};
    else
      \texttt{Elem[result, e, esize]} = \texttt{Extend(data, esize, unsigned)};
  \texttt{Z}[t] = \texttt{result};
LDFF1B (scalar plus scalar)

Contiguous load first-fault unsigned bytes to vector (scalar index).

Contiguous load with first-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>dtype&lt;3:1&gt; dtype&lt;0&gt;</td>
</tr>
</tbody>
</table>

LDFF1B { <Zt>.B }, <Pg>/Z, [{Xn|SP}>{, <Xm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>dtype&lt;3:1&gt; dtype&lt;0&gt;</td>
</tr>
</tbody>
</table>

LDFF1B { <Zt>.H }, <Pg>/Z, [{Xn|SP}>{, <Xm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>dtype&lt;3:1&gt; dtype&lt;0&gt;</td>
</tr>
</tbody>
</table>

LDFF1B { <Zt>.S }, <Pg>/Z, [{Xn|SP}>{, <Xm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
64-bit element

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

\[ \text{dtype<3:1>dtype<0>} \]

LDFF1B \{ \text{<Zt>.D }, \text{<Pg>/Z}, [ \text{<Xn|SP>{}}, \text{<Xm>}] \}

if \text{!HaveSVE}() \text{then UNDEFINED;}
integer \text{t = UInt}(Zt);
integer \text{n = UInt}(Rn);
integer \text{m = UInt}(Rm);
integer \text{g = UInt}(Pg);
integer \text{esize = 64};
integer \text{msize = 8};
boolean \text{unsigned = TRUE};

Assembler Symbols

\(<\text{Zt}>\)  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
\(<\text{Pg}>\)  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\(<\text{Xn|SP}>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\(<\text{Xm}>\)  Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
            ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = base + UInt(offset) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
        else
            (data, fault) = (Zeros(msize), FALSE);

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';
        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';
        if unknown then
            if !fault &&
                    ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif
                    ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros();
            else
                // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);

        offset = offset + 1;
Z[t] = result;
LDFF1B (scalar plus vector)

Gather load first-fault unsigned bytes to vector (vector index).

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: **32-bit unpacked unscaled offset**, **32-bit unscaled offset** and **64-bit unscaled offset**

### 32-bit unpacked unscaled offset

![Binary representation]

```plaintext
LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>], <Zm>.D, <mod>
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 32-bit unscaled offset

![Binary representation]

```plaintext
LDFF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>], <Zm>.S, <mod>
```

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

### 64-bit unscaled offset

![Binary representation]

```plaintext
LDFF1B (scalar plus vector)
```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
offset = Z[m];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09.rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDFF1D (vector plus immediate)

Gather load first-fault doublewords to vector (immediate index).

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 0 0 1 0 1 1 0 1 imm5 1 1 1 Pg Zn Zt

msz<1>msz<0> U ff

LDFF1D { <Zt>.D }, <Pg>/<Z, [<Zn>.D{, #<imm>}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LDFF1D (scalar plus scalar)**

Contiguous load first-fault doublewords to vector (scalar index).

Contiguous load with first-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

![Register Encoding](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Zt</td>
</tr>
<tr>
<td>30</td>
<td>Pg</td>
</tr>
<tr>
<td>29</td>
<td>Rn</td>
</tr>
<tr>
<td>28</td>
<td>Zt</td>
</tr>
<tr>
<td>27</td>
<td>Pg</td>
</tr>
<tr>
<td>26</td>
<td>Rn</td>
</tr>
<tr>
<td>25</td>
<td>Zt</td>
</tr>
<tr>
<td>24</td>
<td>Pg</td>
</tr>
<tr>
<td>23</td>
<td>Rn</td>
</tr>
<tr>
<td>22</td>
<td>Zt</td>
</tr>
<tr>
<td>21</td>
<td>Pg</td>
</tr>
<tr>
<td>20</td>
<td>Rn</td>
</tr>
<tr>
<td>19</td>
<td>Zt</td>
</tr>
<tr>
<td>18</td>
<td>Pg</td>
</tr>
<tr>
<td>17</td>
<td>Rn</td>
</tr>
<tr>
<td>16</td>
<td>Zt</td>
</tr>
<tr>
<td>15</td>
<td>Pg</td>
</tr>
<tr>
<td>14</td>
<td>Rn</td>
</tr>
<tr>
<td>13</td>
<td>Zt</td>
</tr>
<tr>
<td>12</td>
<td>Pg</td>
</tr>
<tr>
<td>11</td>
<td>Rn</td>
</tr>
<tr>
<td>10</td>
<td>Zt</td>
</tr>
<tr>
<td>9</td>
<td>Pg</td>
</tr>
<tr>
<td>8</td>
<td>Rn</td>
</tr>
<tr>
<td>7</td>
<td>Zt</td>
</tr>
<tr>
<td>6</td>
<td>Pg</td>
</tr>
<tr>
<td>5</td>
<td>Rn</td>
</tr>
<tr>
<td>4</td>
<td>Zt</td>
</tr>
<tr>
<td>3</td>
<td>Pg</td>
</tr>
<tr>
<td>2</td>
<td>Rn</td>
</tr>
<tr>
<td>1</td>
<td>Zt</td>
</tr>
<tr>
<td>0</td>
<td>Pg</td>
</tr>
</tbody>
</table>

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>\{, <Xm>, LSL #3}\}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONstrained UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault &&
      ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif
      ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  offset = offset + 1;
Z[t] = result;
LDFF1D (scalar plus vector)

Gather load first-fault doublewords to vector (vector index).

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

32-bit unpacked scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 1 xs 1</td>
</tr>
</tbody>
</table>

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;

32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 1 xs 0</td>
</tr>
</tbody>
</table>

LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 1 1 1</td>
</tr>
</tbody>
</table>

LDFF1D (scalar plus vector)
if !HaveSVE() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;

**64-bit unscaled offset**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**Assembler Symbols**

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Zm>` is the name of the offset scalable vector register, encoded in the "Zm" field.
- `<mod>` is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th><code>&lt;mod&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
offset = Z[m];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        end if
    else
        (data, fault) = (Zeros(msize), FALSE);
    end if

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    end if
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        end if
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    end if

Z[t] = result;
LDFF1H (vector plus immediate)

Gather load first-fault unsigned halfwords to vector (immediate index).

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

msz<1>msz<0> U ff

```
LDFF1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}] }
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

msz<1>msz<0> U ff

```
LDFF1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}] }
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else  // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
LDFF1H (scalar plus scalar)

Contiguous load first-fault unsigned halfwords to vector (scalar index).

Contiguous load with first-faulting behavior of unsigned halfwords to elements of a vector register from the memory
address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address.
After each element access the index value is incremented, but the index register is not updated. Inactive elements will
not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: **16-bit element**, **32-bit element** and **64-bit element**.

### 16-bit element

![16-bit element diagram]

```assembly
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
```

### 32-bit element

![32-bit element diagram]

```assembly
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
```

### 64-bit element

![64-bit element diagram]

```assembly
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
```
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
      (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';

  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault &&
      ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif
      ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  offset = offset + 1;

Z[t] = result;
LDFF1H (scalar plus vector)

Gather load first-fault unsigned halfwords to vector (vector index).

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: **32-bit scaled offset**, **32-bit unpacked scaled offset**, **32-bit unpacked unscaled offset**, **32-bit unscaled offset**, **64-bit scaled offset** and **64-bit unscaled offset**

### 32-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | Zm | 0  | 1  | 1  | Pg | Rn | Zt | U f f |

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

### 32-bit unpacked scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | Zm | 0  | 1  | 1  | Pg | Rn | Zt | U f f |

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;

### 32-bit unpacked unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | Zm | 0  | 1  | 1  | Pg | Rn | Zt | U f f | msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

64-bit scaled offset

LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

64-bit unscaled offset
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(PRO) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09.rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LDFF1SB (vector plus immediate)**

Gather load first-fault signed bytes to vector (immediate index).

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0</td>
<td>0 0 1 imm5</td>
<td>1 0 1 Pg</td>
<td>Zn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Zt</td>
</tr>
</tbody>
</table>
```

LDFF1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### 64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8</th>
<th>7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0</td>
<td>0 0 1 imm5</td>
<td>1 0 1 Pg</td>
<td>Zn</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Zt</td>
</tr>
</tbody>
</table>
```

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### Assembler Symbols

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the base scalable vector register, encoded in the "Zn" field.
- **<imm>** Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bites(VL) base = Z[n];
bites(64) addr;
bites(PL) mask = P[g];
bites(VL) result;
bites(VL) orig = Z[t];
bites(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
**LDFF1SB (scalar plus scalar)**

Contiguous load first-fault signed bytes to vector (scalar index).

Contiguous load with first-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

### 16-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;

### 32-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;

### 64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm}>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

```pseudocode
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        base = SP[];
    else
        base = X[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = base + UInt(offset) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
        else
            (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault &&
            ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif
            ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else
            // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    offset = offset + 1;
Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDFF1SB (scalar plus vector)

Gather load first-fault signed bytes to vector (vector index).

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset

### 32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25</th>
<th>24 23 22 21 20 19 18 17 16 15 14 13 12 11 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0</td>
<td>0 0 xs 0 Zm 0 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25</th>
<th>24 23 22 21 20 19 18 17 16 15 14 13 12 11 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0</td>
<td>0 0 xs 0 Zm 0 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDFF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25</th>
<th>24 23 22 21 20 19 18 17 16 15 14 13 12 11 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0</td>
<td>0 0 1 0 Zm 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDFF1SB (scalar plus vector)
LDFF1SB \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D] \\

if !HaveSVE() then UNDEFINED; 
integer t = UInt(Zt); 
integer n = UInt(Rn); 
integer m = UInt(Zm); 
integer g = UInt(Pg); 
integer esize = 64; 
integer msize = 8; 
integer offs_size = 64; 
boolean unsigned = FALSE; 
boolean offs_unsigned = TRUE; 
integer scale = 0;

### Assembler Symbols

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<Zm>** Is the name of the offset scalable vector register, encoded in the "Zm" field.
- **<mod>** Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
offset = Z[m];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize] < offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
```
LDFF1SH (vector plus immediate)

Gather load first-fault signed halfwords to vector (immediate index).

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
| msz<1>msz<0> | U | ff |

LDFF1SH {<Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

### 64-bit element

|    | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
| msz<1>msz<0> | U | ff |

LDFF1SH {<Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

**Assembler Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ELEM[mask, e, esize] == '1' then
    addr = ZeroExtend(ELEM[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccTypeNORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccTypeNONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);

  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';

  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault && ConstrainsUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainsUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);

  Z[t] = result;
**LDFF1SH (scalar plus scalar)**

Contiguous load first-fault signed halfwords to vector (scalar index).

Contiguous load with first-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

### 32-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>
```

```
LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
```

### 64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>
```

```
LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
```

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
```

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bis(msize) data;
bits(64) offset = X[m];

constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        base = SP[];
    else
        base = X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = base + UInt(offset) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
        else
            (data, fault) = (Zeros(msize), FALSE);

        // FFR elements set to FALSE following a suppressed access/fault
        faulted = faulted || fault;
        if faulted then
            ElemFFR[e, esize] = '0';

        // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
        unknown = unknown || ElemFFR[e, esize] == '0';

        if unknown then
            if !fault &&
                ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif
                ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros();
            else
                // merge
                Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);

        offset = offset + 1;

    Z[t] = result;
LDFF1SH (scalar plus vector)

Gather load first-fault signed halfwords to vector (vector index).

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 0 0 0 0 1 0 0 1 xs 1 Zm 0 0 1 Pg Rn Zt
```

```
LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]
```

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

32-bit unpacked scaled offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 0 1 xs 1 Zm 0 0 1 Pg Rn Zt
```

```
LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
```

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

32-bit unpacked unscaled offset

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 1 1 0 0 0 1 0 0 1 xs 0 Zm 0 0 1 Pg Rn Zt
```

```
LDFF1SH (scalar plus vector)
Page 2007
```
LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>xs</td>
<td>0</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td>U</td>
<td>ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td>U</td>
<td>ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td>U</td>
<td>ff</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0>
LDFF1SH \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

**Assembler Symbols**

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<Zm>** Is the name of the offset scalable vector register, encoded in the "Zm" field.
- **<mod>** Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(VL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = Z[m];
for e = 0 to elements-1
  ifElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONstrained Unpredictable after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    else
      Elem[result, e, esize] = Extend(data, esize, unsigned);
  Z[t] = result;
```
**LDFF1SW (vector plus immediate)**

Gather load first-fault signed words to vector (immediate index).

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th>Index</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>msz&lt;1&gt; msz&lt;0&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Uff</td>
<td></td>
</tr>
</tbody>
</table>

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D, #<imm>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);

**Assembler Symbols**

- `<Zt>` is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the base scalable vector register, encoded in the "Zn" field.
- `<imm>` is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bite(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
    // Mem[] will not return if a fault is detected for the first active element
        data = Mem[addr, mbytes, AccType_NORMAL];
        first = FALSE;
    else
    // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    end
    (data, fault) = (Zeros(msize), FALSE);
// FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrantUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrantUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        end
    end
    Elem[result, e, esize] = Elem[orig, e, esize];
end
Z[t] = result;
LDFF1SW (scalar plus scalar)

Contiguous load first-fault signed words to vector (scalar index).

Contiguous load with first-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

dtype<3:1> dtype<0>

LDFF1SW {<Zt>.D}, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

if !HaveSVE() then UNDEFINED;
integer t = Uint(Zt);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer g = Uint(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  offset = offset + 1;
Z[t] = result;
LDFF1SW (scalar plus vector)

Gather load first-fault signed words to vector (vector index).

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset, and 64-bit unscaled offset.

32-bit unpacked scaled offset

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

if ![HaveSVE()] then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked unscaled offset

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if ![HaveSVE()] then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]
LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  |

LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm>  Is the name of the offset scalable vector register, encoded in the "Rn" field.
<mod> Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
offset = Z[m];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_NORMAL];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
        else
            (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a supressed access/fault
        faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault &&
            ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif
            ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else
            // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        else
            Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
LDFF1W (vector plus immediate)

Gather load first-fault unsigned words to vector (immediate index).

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

```
LDFF1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element

```
LDFF1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
      (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
      ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
      if !fault && ConstraintUnpredictableBool(Unpredictable_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
      elsif ConstraintUnpredictableBool(Unpredictable_SVELDNFZERO) then
        Elem[result, e, esize] = Zeros();
      else
        Elem[result, e, esize] = Elem[orig, e, esize];
      else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    Z[t] = result;
LDFF1W (scalar plus scalar)

Contiguous load first-fault unsigned words to vector (scalar index).

Contiguous load with first-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 |

```plaintext
LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, < Xm>, LSL #2}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;

64-bit element

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 |

```plaintext
LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, < Xm>, LSL #2}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[ ];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + UInt(offset) * mbytes;
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
  else
    (data, fault) = (Zeros(msize), FALSE);

  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault &&
      ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif
      ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  offset = offset + 1;
Z[t] = result;
LDFF1W (scalar plus vector)

Gather load first-fault unsigned words to vector (vector index).

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>xs</td>
<td>1</td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked scaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>xs</td>
<td>1</td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;

32-bit unpacked unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>xs</td>
<td>0</td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0> U ff
LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | xs | 0  | Zm | 0  | 1  | 1  | Pg | Rn | Zt |

LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | Zm | 1  | 1  | 1  | Pg | Rn | Zt |

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| msz<1>|msz<0>|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | U ff |
LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) offset;
b bits(PL) mask = P[g];
b bits(VL) result;
b bits(VL) orig = Z[t];
b bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
  offset = Z[m];
for e = 0 to elements-1
  if Elemp[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    if first then
      // Mem[] will not return if a fault is detected for the first active element
      data = Mem[addr, mbytes, AccType_NORMAL];
      first = FALSE;
    else
      // MemNF[] will return fault=TRUE if access is not performed for any reason
      (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    end
  else
    (data, fault) = (Zeros(msize), FALSE);
  end
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || Elemp[e, esize] == '0';
  if unknown then
    if !fault &&
      ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif
      ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
        Elem[result, e, esize] = Zeros();
      end
    else
      // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
    end
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  end
Z[t] = result;
LDNF1B

Contiguous load non-fault unsigned bytes to vector (immediate index).

Contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 4 classes: 8-bit element, 16-bit element, 32-bit element and 64-bit element

8-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21 20 19 18 17 16 15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 0 0 0 1 1 1</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LDNF1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21 20 19 18 17 16 15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 0 1 1 1 1</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LDNF1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22</th>
<th>21 20 19 18 17 16 15 14 13 12</th>
<th>11 10 9 8 7 6 5 4 3 2</th>
<th>1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 0 0 1 0 0 0 1 0</td>
<td>imm4</td>
<td>1 0 1</td>
<td>Pg</td>
</tr>
</tbody>
</table>

dtype<3:1>dtype<0>

LDNF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | imm4 | 1 | 0 | 1 | Pg | Rn | Zt |

dtype<3:1>dtype<0>

LDNF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
ninteger esize = 64;
ninteger msize = 8;
boolean unsigned = TRUE;
ninteger offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
    if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros();
        else // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    addr = addr + mbytes;
Z[t] = result;
```
LDNF1D

Contiguous load non-fault doublewords to vector (immediate index).

Contiguous load with non-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

LDNF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bv<64> orig = Z[t];
bv<64> data;

const integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if n == 31 then
    if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1'
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

// FFR elements set to FALSE following a supressed access/fault
faulted = faulted || fault;
if faulted then
    ElemFFR[e, esize] = '0';

// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
    if !fault & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA)
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO)
        Elem[result, e, esize] = Zeros();
    else
        // merge
        Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

addr = addr + mbytes;

Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNF1H

Contiguous load non-fault unsigned halfwords to vector (immediate index).

Contiguous load with non-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 1 imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>
```

LDNF1H { <Zt>.H }, <Pg>/Z, [ <Xn|SP>{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

32-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 1 imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>
```

LDNF1H { <Zt>.S }, <Pg>/Z, [ <Xn|SP>{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

64-bit element

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 1 imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>
```

LDNF1H { <Zt>.D }, <Pg>/Z, [ <Xn|SP>{, #imm}, MUL VL]}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
### Assembler Symbols

- **<Zt>** Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<imm>** Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
    if Constr
```
LDNF1SB

Contiguous load non-fault signed bytes to vector (immediate index).

Contiguous load with non-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 3 classes: 16-bit element, 32-bit element and 64-bit element

16-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 1 0 1 0 1 imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDNF1SB { <Zt> .H }, <Pg> / Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 0 1 1 imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDNF1SB { <Zt> .S }, <Pg> / Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0 1 1 0 0 1 imm4 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

LDNF1SB { <Zt> .D }, <Pg> / Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
  if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
// FFR elements set to FALSE following a suppressed access/fault
faulted = faulted || fault;
if faulted then
  ElemFFR[e, esize] = '0';
// Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
unknown = unknown || ElemFFR[e, esize] == '0';
if unknown then
  if fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
    Elem[result, e, esize] = Zeros();
  else // merge
    Elem[result, e, esize] = Elem[orig, e, esize];
else
  Elem[result, e, esize] = Extend(data, esize, unsigned);
addr = addr + mbytes;
Z[t] = result;
LDNF1SH

Contiguous load non-fault signed halfwords to vector (immediate index).

Contiguous load with non-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: **32-bit element** and **64-bit element**

32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

LDNF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

LDNF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if n == 31 then
    if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);
    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if fault then
            ElemP[mask, e, esize] == '1' then
                // MemNF[] will return fault=TRUE if access is not performed for any reason
                (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
            else
                (data, fault) = (Zeros(msize), FALSE);
            // FFR elements set to FALSE following a supressed access/fault
            faulted = faulted || fault;
            if faulted then
                ElemFFR[e, esize] = '0';
            // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
            unknown = unknown || ElemFFR[e, esize] == '0';
            if unknown then
                if fault then
                    ElemP[mask, e, esize] == '1' then
                        ElemP(result, e, esize) = Extend(data, esize, unsigned);
                    else
                        ElemP(result, e, esize) = Zeros();
                    else
                        // merge
                        ElemP(result, e, esize) = ElemP(orig, e, esize);
                    else
                        ElemP(result, e, esize) = Extend(data, esize, unsigned);
                    addr = addr + mbytes;
                Z[t] = result;
LDNF1SW

Contiguous load non-fault signed words to vector (immediate index).

Contiguous load with non-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
| 1 0 1 0 0 1 0 | 0 1 0 | 1 | imm4 | 1 0 | Pg | Rn | Zt |
```

dtype<3:1>dtype<0>

LDNF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

- `<Zt>` : Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` : Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` : Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` : Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
  if constrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);

  // FFR elements set to FALSE following a suppressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';

  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFR[e, esize] == '0';
  if unknown then
    if !fault & constrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif constrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);

addr = addr + mbytes;
Z[t] = result;
Contiguous load non-fault unsigned words to vector (immediate index).

Contiguous load with non-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

![32-bit element pattern](image)

LDNF1W {<Zt>.S}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

### 64-bit element

![64-bit element pattern](image)

LDNF1W {<Zt>.D}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(VL) orig = Z[t];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if n == 31 then
  if ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    // MemNF[] will return fault=TRUE if access is not performed for any reason
    (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
  else
    (data, fault) = (Zeros(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  if faulted then
    ElemFFF[e, esize] = '0';
  if faulted then
    Elem[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || ElemFFF[e, esize] == '0';
  if unknown then
    if !fault & & ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
      Elem[result, e, esize] = Zeros();
    else // merge
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  addr = addr + mbytes;
Z[t] = result;
```
LDNT1B (vector plus scalar)

Gather load non-temporal unsigned bytes.

Gather load non-temporal of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

**32-bit unscaled offset**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | | | | | | | | | | | | | | | | | | | |

LDNT1B {<Zt>.S}, <Pg>/Z, [<Zn>.S{}<Xm>]

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = TRUE;

**64-bit unscaled offset**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | | | | | | | | | | | | | | | | | | | |

LDNT1B {<Zt>.D}, <Pg>/Z, [<Zn>.D{}<Xm>]

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = TRUE;

**Assembler Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_STREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
**LDNT1B (scalar plus immediate)**

Contiguous load non-temporal bytes to vector (immediate index).

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 imm4 1 1 1 Pg Rn Zt
```

```plaintext
LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

**Assembler Symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bites(64) base;
bites(64) addr;
bites(PL) mask = P[g];
bites(VL) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSفاءignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
    addr = base + offset * elements * mbytes;
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
        else
            Elem[result, e, esize] = Zeros();
        addr = addr + mbytes;
Z[t] = result;
```
Contiguous load non-temporal bytes to vector (scalar index).

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0

msz<1>msz<0>

LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '1111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
     ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = X[m];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
  else
    Elem[result, e, esize] = Zeros();
  offset = offset + 1;
Z[t] = result;
LDNT1D (vector plus scalar)

Gather load non-temporal unsigned doublewords.

Gather load non-temporal of doublewords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL / esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize / 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_STREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
```
LDNT1D (scalar plus immediate)

Contiguous load non-temporal doublewords to vector (immediate index).

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

- **msz<1>msz<0>**
- LDNT1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

### Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

### Operation

- **CheckSVEEnabled();**
- integer elements = VL DIV esize;
- bits(64) base;
- bits(64) addr;
- bits(VL) mask = P[g];
- bits(VL) result;
- constant integer mbytes = esize DIV 8;
- if n == 31 then
  - if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
  else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
- addr = base + offset * elements * mbytes;
- for e = 0 to elements-1
  - if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
  else
    Elem[result, e, esize] = Zeros();
  addr = addr + mbytes;
- Z[t] = result;
LDNT1D (scalar plus scalar)

Contiguous load non-temporal doublewords to vector (scalar index).

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 1 0 0 1 0 1 1 0 0 | Rm | 1 1 0 | Pg | Rn | Zt |

\[\text{msz<1>msz<0>}\]

LDNT1D \{<Zt>.D\}, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

if \(!\text{HaveSVE}()\) then UNDEFINED;
if \(\text{Rm} == '1111'\) then UNDEFINED;
integer \(t = \text{UInt}(Zt)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
integer \(g = \text{UInt}(Pg)\);
integer \(\text{esize} = 64\);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = \(\text{VL} \div \text{esize}\);
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = \(P[g]\);
bits(VL) result;
constant integer mbytes = \(\text{esize} \div 8\);
if \(!\text{HaveMTEExt}()\) then SetTagCheckedInstruction(TRUE);
if \(n == 31\) then
  if \(!\text{LastActiveElement}(\text{mask}, \text{esize}) >= 0 ||\)
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
  base = \(\text{SP}[]\);
else
  base = \(\text{X}[n]\);
offset = \(\text{X}[m]\);
for \(e = 0\) to elements-1
  addr = base + \text{UInt}(offset) * mbytes;
  if \(!\text{Elem}(\text{mask}, e, \text{esize}) == '1'\) then
    \(\text{Elem}(\text{result, e, esize}) = \text{Mem}(\text{addr, mbytes, AccType_STREAM})\);
  else
    \(\text{Elem}(\text{result, e, esize}) = \text{Zeros}()\);
offset = offset + 1;
\(Z[t] = \text{result}\);
LDNT1H (vector plus scalar)

Gather load non-temporal unsigned halfwords.

Gather load non-temporal of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

### 32-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>msz&lt;1&gt;msz&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0 0 0 1 0 0</td>
<td>Rm 1 0 1</td>
</tr>
<tr>
<td></td>
<td>Pg Zn Zt</td>
</tr>
</tbody>
</table>

LDNT1H \{ <Zt>.S \}, <Pg>/Z, \[<Zn>.S\}, <Xm>\}

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = TRUE;

### 64-bit unscaled offset

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>msz&lt;1&gt;msz&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1 0 0 1 0 0</td>
<td>Rm 1 1 0</td>
</tr>
<tr>
<td></td>
<td>Pg Zn Zt</td>
</tr>
</tbody>
</table>

LDNT1H \{ <Zt>.D \}, <Pg>/Z, \[<Zn>.D\}, <Xm>\}

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = TRUE;

### Assembler Symbols

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
in integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_STREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
**LDNT1H (scalar plus immediate)**

Contiguous load non-temporal halfwords to vector (immediate index).

Contiguous load non-temporal halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

![Bit field diagram]

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| msz<1> | msz<0> |

LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(VL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
else
    addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;

Z[t] = result;
LDNT1H (scalar plus scalar)

Contiguous load non-temporal halfwords to vector (scalar index).

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n];
offset = X[m];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
    else
        Elem[result, e, esize] = Zeros();
    offset = offset + 1;
Z[t] = result;
```

LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;

```
**LDNT1SB**

Gather load non-temporal signed bytes.

Gather load non-temporal of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector. A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

### 32-bit unscaled offset

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

LDNT1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>})

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
boolean unsigned = FALSE;

### 64-bit unscaled offset

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

LDNT1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>})

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;

**Assembler Symbols**

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_STREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
LDNT1SH

Gather load non-temporal signed halfwords.

Gather load non-temporal of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |
| Rm | Pg | Zn | Zt |

LDNT1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
boolean unsigned = FALSE;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |
| Rm | Pg | Zn | Zt |

LDNT1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm}>]

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
boolean unsigned = FALSE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
native elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_STREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();

Z[t] = result;
LDNT1SW

Gather load non-temporal signed words.

Gather load non-temporal of signed words to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1'
        addr = ZeroExtend(Elem(base, e, esize), 64) + offset;
        data = Mem[addr, mbytes, AccType_STREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros();
Z[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNT1W (vector plus scalar)

Gather load non-temporal unsigned words.

Gather load non-temporal of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

LDNT1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

if !HaveSVE2() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
boolean unsigned = TRUE;

64-bit unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

LDNT1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

if !HaveSVE2() then UNDEFINED;

integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
boolean unsigned = TRUE;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    data = Mem[addr, mbytes, AccType_STREAM];
    Elem[result, e, esize] = Extend(data, esize, unsigned);
  else
    Elem[result, e, esize] = Zeros();
Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LDNT1W (scalar plus immediate)**

Contiguous load non-temporal words to vector (immediate index).

Contiguous load non-temporal of words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Rn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MSZ<1> MSZ<0>

LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[0];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
    else
        Elem[result, e, esize] = Zeros();
    addr = addr + mbytes;
Z[t] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
LDNT1W (scalar plus scalar)

Contiguous load non-temporal words to vector (scalar index).

Contiguous load non-temporal words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

msz<1>msz<0>

LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
icnter elements = VL DIV esize;
bixts(64) base;
bixts(64) addr;
bixts(64) offset;
bixts(VL) mask = P[g];
bixts(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
        base = SP[];
    else
        base = X[n];
offset = X[m];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_STREAM];
    else
        Elem[result, e, esize] = Zeros();
offset = offset + 1;
Z[t] = result;
LDR (predicate)

Load predicate register.

Load a predicate register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated.

The load is performed as a stream of bytes containing 8 consecutive predicate bits in ascending element order, without any endian conversion.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | imm9h | 0  | 0  | 0  | imm9l | Rn | 0  | Pt |

LDR <Pt>, [<Xn|SP>\{, #<imm>, MUL VL}\]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler Symbols

<Pt>  Is the name of the destination scalable predicate register, encoded in the "Pt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>  Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
integer elements = PL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(PL) result;
if n == 31 then
  CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_NORMAL, FALSE);
for e = 0 to elements-1
  Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_NORMAL, aligned];
  offset = offset + 1;

P[t] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDR (vector)

Load vector register.

Load a vector register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The load is performed as a stream of byte elements in ascending element order, without any endian conversion.

```
LDR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Asmber Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(VL) result;
if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_NORMAL, FALSE);
for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_NORMAL, aligned];
    offset = offset + 1;
Z[t] = result;
```
**LSL (immediate, predicated)**

Logical shift left by immediate (predicated).

Shift left by immediate each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>tzh</th>
<th>tszl</th>
<th>const</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1 1 1 0 0</td>
</tr>
</tbody>
</table>

LSL \(<\text{Zdn}>.\langle T\rangle, <\text{Pg}>/M, <\text{Zdn}>.\langle T\rangle, \#<\text{const}>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
bits(4) tsize = tzh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = \(\text{UInt}(\text{Pg})\);
integer dn = \(\text{UInt}(\text{Zdn})\);
integer shift = \(\text{UInt}(\text{tsize}:\text{imm3}) - \text{esize}\);

**Assembler Symbols**

\(<\text{Zdn}>\) Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

\(<\text{T}>\) Is the size specifier, encoded in “tzh:tszl”:

<table>
<thead>
<tr>
<th>tzh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{const}>\) Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3”.

**Operation**

\(\text{CheckSVEEnabled}()\);
integer elements = \(\text{VL} \div \text{esize}\);
bits(\text{VL}) operand1 = \(\text{Z}[\text{dn}]\);
bits(\text{PL}) mask = \(\text{P}[g]\);
bits(\text{VL}) result;
for e = 0 to elements-1
  bits(e) element1 = \(\text{Elem}[\text{operand1}, e, \text{esize}]\);
  if \(\text{ElemP}[\text{mask, e, esize}] == '1'\) then
    \(\text{Elem}[\text{result, e, esize}] = \text{LSL}(\text{element1, shift})\);
  else
    \(\text{Elem}[\text{result, e, esize}] = \text{Elem}[\text{operand1, e, esize}]\);
\(\text{Z}[\text{dn}] = \text{result}\);

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
• The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
• The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
• The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSL (wide elements, predicated)

Logical shift left by 64-bit wide elements (predicated).

Shift left active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | Pg | Zm | Zdn|

LSL <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = LSL(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
    Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
  • The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSL (vectors)

Logical shift left by vector (predicated).

Shift left active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-------------|-------------|
| 0 0 0 0 0 1 0 0 | 0 1 0 0 | 1 1 1 0 0 | Pg | Zm | Zdn |
```

LSL \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>\)

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 \(<\text{UInt}(\text{size});
integer g = \text{UInt}(Pg);
integer dn = \text{UInt}(Zdn);
integer m = \text{UInt}(Zm);
```

Assembler Symbols

\(<Zdn>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = \text{VL} / \text{esize};
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element2), esize);
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = LSL(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**LSL (immediate, unpredicated)**

Logical shift left by immediate (unpredicated).

Shift left by immediate each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

```
  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  tszh | tszl | imm3  1  0  0  1  1  1  Zn  | Zd
```

LSL <Zd>.<T>, <Zn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “tszh:tszl”:
  - tszh  tszl  <T>
  - 00  00  RESERVED
  - 00  01  B
  - 00  1x  H
  - 01  xx  S
  - 1x  xx  D
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
- `<const>` Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = LSL(element1, shift);
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSL (wide elements, unpredicated)

Logical shift left by 64-bit wide elements (unpredicated).

Shift left all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{cccccccccccccccccccccccc}
\end{array}
\]

| 0 | 0 | 0 | 0 | 1 | 0 | 0 | size | 1 | Zm | 1 | 0 | 0 | 1 | 1 | Zn | Zd |

\[
\text{LSL } <\text{Zd}> .<\text{T}> , <\text{Zn}> .<\text{T}> , <\text{Zm}> .D
\]

if \(!\text{HaveSVE}()\) then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 \ll \text{UInt}(size);
integer n = \text{UInt}(Zn);
integer m = \text{UInt}(Zm);
integer d = \text{UInt}(Zd);

**Assembler Symbols**

- \(<\text{Zd}>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.
- \(<\text{T}>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>(\text{size})</th>
<th>(&lt;\text{T}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- \(<\text{Zn}>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.
- \(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

\[
\text{CheckSVEEnabled();}
\]

integer elements = \text{VL} \div esize;
bits(\text{VL}) \text{operand1} = Z[n];
bits(\text{VL}) \text{operand2} = Z[m];
bits(\text{VL}) \text{result};

for e = 0 to elements-1
  bits(esize) \text{element1} = \text{Elem}[\text{operand1}, e, esize];
  bits(64) \text{element2} = \text{Elem}[\text{operand2}, (e * esize) \div 64, 64];
  integer \text{shift} = \text{Min}(\text{UInt}(\text{element2}), esize);
  \text{Elem}[\text{result}, e, esize] = \text{LSL}(\text{element1}, \text{shift});
\]

\(Z[d] = \text{result};\)

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**LSLR**

Reversed logical shift left by vector (predicated).

Reversed shift left active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| 0 0 0 0 0 1 0 0 | size | 0 1 0 | 1 | 1 | 0 0 | Pg | Zm | Zdn |
| R | L | U |

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
    integer shift = Min(UInt(element1), esize);
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = LSL(element2, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSR (immediate, predicated)

Logical shift right by immediate (predicated).

Shift right by immediate, inserting zeroes, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 0 tszh 0 0 0 0 1 1 0 0 Pg tszl imm3 Zdn |

LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**LSR (wide elements, predicated)**

Logical shift right by 64-bit wide elements (predicated).

Shift right, inserting zeroes, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer esize = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
  integer shift = Min(UInt(element2), esize);
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSR(element1, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
  • The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
LSR (vectors)

Logical shift right by vector (predicated).

Shift right, inserting zeroes, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

```
0 0 0 0 0 1 0 0 | size | 0 1 0 0 1 1 0 0 | Pg | Zm | Zdn
R L U
```

\[\text{LSR } (<\text{Zdn}>, <\text{T}>, <\text{Pg}> / M, <\text{Zdn}>, <\text{T}>, <\text{Zm}>, <\text{T}>)\]

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer \(\text{esize} = 8 \ll \text{UInt}(\text{size})\);
integer \(g = \text{UInt}(\text{Pg})\);
integer \(dn = \text{UInt}(\text{Zdn})\);
integer \(m = \text{UInt}(\text{Zm})\);

**Assembler Symbols**

- `<\text{Zdn}>`  Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<\text{T}>`  Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\text{T}&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<\text{Pg}>`  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<\text{Zm}>`  Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

\[\text{CheckSVEEnabled}()\];
integer \(\text{elements} = \text{VL} \div \text{esize}\);
bits(\text{PL}) \text{ mask} = P[g];
bits(\text{VL}) \text{ operand1} = Z[dn];
bits(\text{VL}) \text{ operand2} = Z[m];
bits(\text{VL}) \text{ result};

for \(e = 0\) to elements-1
  bits(esize) \text{ element1} = Elem[operand1, e, esize];
  bits(esize) \text{ element2} = Elem[operand2, e, esize];
  integer \text{ shift} = \text{Min}(\text{UInt}(\text{element2}), \text{esize});
  if \(\text{ElemP}[\text{mask}, e, \text{esize}] \equiv '1'\) then
    \(\text{Elem[result, e, esize]} = \text{LSR} (\text{element1}, \text{shift})\);
  else
    \(\text{Elem[result, e, esize]} = \text{Elem[operand1, e, esize]}\);
\(Z[dn] = \text{result};\)

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**LSR (immediate, unpredicated)**

Logical shift right by immediate (unpredicated).

Shift right by immediate, inserting zeroes, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
   0  0  0  0  1  0  0  tszh | tszl | imm3 | 1  0  0  1  0  1 | Zn | Zd
```

**LSR <Zd>.<T>, <Zn>.<T>, #<const>**

if ![HaveSVE]() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsz:imm3);

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th></th>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tszh:imm3”.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  Elem[result, e, esize] = LSR(element1, shift);
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSR (wide elements, unpredicated)

Logical shift right by 64-bit wide elements (unpredicated).

Shift right, inserting zeroes, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.

```
0 0 0 0 0 0 1 0 0 | size | 1 |
Zm 1 0 0 0 0 0 1 | Zn | Zd
```

LSR <Zd>.<T>, <Zn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
  integer shift = Min(UInt(element2), esize);
  Elem[result, e, esize] = LSR(element1, shift);
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
LSRR

Reversed logical shift right by vector (predicated).

Reversed shift right, inserting zeroes, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|**size** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| **Pg**  |   |   |   |   |   |   | R | L | U |
| **Zm**  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| **Zdn** |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

LSRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```cpp
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler Symbols**

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in “size”:

  ```markdown
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
  ```

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```cpp
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  integer shift = Min(UInt(element1), esize);
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = LSR(element2, shift);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Multiply-add vectors (predicated), writing multiplicand \( Z_{dn} = Z_a + Z_{dn} \times Z_{m} \).

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{ccccccccccccc}
\end{array}
\]

\[
\begin{array}{ccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \text{size} & 0 & Z_m & 1 & 1 & 0 & Pg & Z_a & Z_{dn} & \text{op}
\end{array}
\]

\text{MAD} \langle Z_{dn}\rangle, \langle T\rangle, \langle Pg\rangle/M, \langle Z_m\rangle, \langle T\rangle, \langle Z_a\rangle, \langle T\rangle

\begin{verbatim}
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Z_{dn});
integer m = UInt(Z_{m});
integer a = UInt(Z_{a});
boolean sub_op = FALSE;
\end{verbatim}

**Assembler Symbols**

\(<Z_{dn}>\) Is the name of the first source and destination scalable vector register, encoded in the "Z_{dn}" field.

\(<T>\) Is the size specifier, encoded in "size":

\[
\begin{array}{c|c}
\text{size} & <T> \\
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\end{array}
\]

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Z_{m}>\) Is the name of the second source scalable vector register, encoded in the "Z_{m}" field.

\(<Z_{a}>\) Is the name of the third source scalable vector register, encoded in the "Z_{a}" field.

**Operation**

\text{CheckSVEEnabled();}

integer elements = \text{VL} \div \text{esize};

\text{bits(PL)} mask = P[g];

\text{bits(VL)} operand1 = Z[dn];

\text{bits(VL)} operand2 = Z[m];

\text{bits(VL)} operand3 = Z[a];

\text{bits(VL)} result;

for e = 0 to elements-1

\text{integer element1} = \text{UInt(Elem[operand1, e, esize]);}

\text{integer element2} = \text{UInt(Elem[operand2, e, esize]);}

\text{if} \text{Elem[mask, e, esize]} == '1' \text{then}

\text{integer product} = \text{element1} \times \text{element2};

\text{if sub_op then}

\text{Elem[result, e, esize]} = \text{Elem[operand3, e, esize]} - \text{product};

\text{else}

\text{Elem[result, e, esize]} = \text{Elem[operand3, e, esize]} + \text{product};

\text{else}

\text{Elem[result, e, esize]} = \text{Elem[operand1, e, esize]};

Z[dn] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MATCH

Detect any matching elements, setting the condition flags.

This instruction compares each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. Where the first source element detects any matching characters in the second segment it places true in the corresponding element of the destination predicate, otherwise false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Assembler Symbols

- **<Pd>** is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<T>** is the size specifier, encoded in “size<0>“:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
</tbody>
</table>
- **<Pg>** is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Zm>** is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```pseudo
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(P) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(P) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer segmentbase = e - (e MOD eltspersegment);
    ElemP[result, e, esize] = '0';
    bits(esize) element1 = Elem[operand1, e, esize];
    for i = segmentbase to segmentbase + eltspersegment - 1
      bits(esize) element2 = Elem[operand2, i, esize];
      if element1 == element2 then
        ElemP[result, e, esize] = '1';
      else
        ElemP[result, e, esize] = '0';
  else
    ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
MLA (vectors)

Multiply-add vectors (predicated), writing addend $Z_{da} = Z_{da} + Z_{n} \times Z_{m}$.

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

```
0 0 0 0 0 1 0 0 | size | 0 1 0 | Pg | Zn | Zda
```

MLA $<Z_{da}> .<T>$, $<Pg>/M, <Zn>.<T>$, $<Zm>.<T>$

if !HaveSVE() then UNDEFINED;
integer esize = 8 $\ll$ UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean sub_op = FALSE;

Assembler Symbols

$<Z_{da}>$ Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

$<T>$ Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$<Pg>$ Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

$<Zn>$ Is the name of the first source scalable vector register, encoded in the "Zn" field.

$<Zm>$ Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bis(VL) operand1 = Z[n];
bis(VL) operand2 = Z[m];
bis(VL) operand3 = Z[da];
bis(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer product = element1 * element2;
        if sub_op then
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        else
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MLA (indexed)

Multiply-add to accumulator (indexed).

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added to the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: **16-bit**, **32-bit** and **64-bit**

### 16-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Zm | 0  | 0  | 0  | 0  | 1  | 0  | Zn | Zda |


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | Zm | 0  | 0  | 0  | 0  | 1  | 0  | Zn | Zda |

MLA <Zda>.S, <Zn>.S, <Zm>.S<imm>

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

### 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | i1 | Zm | 0  | 0  | 0  | 0  | 1  | 0  | Zn | Zda |


if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

### Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
Is the name of the first source scalable vector register, encoded in the "Zn" field.

For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MLS (vectors)

Multiply-subtract vectors (predicated), writing addend \([Zda = Zda - Zn \times Zm]\).

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| 0 0 0 0 0 1 0 0 | size | 0 | Zm | 0 | 1 | Pg | Zn | Zda |
```

MLS \(<Zda>\).

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 \(<\text{UInt}(\text{size})\);
integer g = \text{UInt}(Pg);
integer n = \text{UInt}(Zn);
integer m = \text{UInt}(Zm);
integer da = \text{UInt}(Zda);
boolean sub_op = TRUE;

Assembler Symbols

- \(<Zda>\): Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- \(<T>\): Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<Pg>\): Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<Zn>\): Is the name of the first source scalable vector register, encoded in the "Zn" field.
- \(<Zm>\): Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = \text{VL} \text{DIV} esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = \text{UInt}(\text{Elem}[operand1, e, esize]);
    integer element2 = \text{UInt}(\text{Elem}[operand2, e, esize]);
    if \text{Elem}[mask, e, esize] == '1' then
        integer product = element1 * element2;
        if sub_op then
            \text{Elem}[result, e, esize] = \text{Elem}[operand3, e, esize] - product;
        else
            \text{Elem}[result, e, esize] = \text{Elem}[operand3, e, esize] + product;
    else
        \text{Elem}[result, e, esize] = \text{Elem}[operand3, e, esize];
Z[da] = result;
```
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MLS (indexed)

Multiply-subtract from accumulator (indexed).

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted from the corresponding elements of the addend and destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: 16-bit, 32-bit and 64-bit

16-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 | i3h | i3l | Zm | 0 0 0 0 1 1 | Zn | Zda


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

32-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 | 1 | 0 | i2 | Zm | 0 0 0 0 1 1 | Zn | Zda

MLS <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 | 1 | 1 | i1 | Zm | 0 0 0 0 1 1 | Zn | Zda


if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
Is the name of the first source scalable vector register, encoded in the "Zn" field.

For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

---

**Operation**

```pseudocode
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;
Z[da] = result;
```

---

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV (predicate, predicated, zeroing)

Move predicates (zeroing).

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of AND, ANDS (predicates). This means:

- The encodings in this description are named to match the encodings of AND, ANDS (predicates).
- The description of AND, ANDS (predicates) gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Pm | 0 | 1 | Pg | 0 | Pn | 0 | Pd

MOV <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to


and is the preferred disassembly when S == '0' && Pn == Pm.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of AND, ANDS (predicates) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
MOV (immediate, predicated, zeroing)

Move signed integer immediate to vector elements (zeroing).

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of CPY (immediate, zeroing). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, zeroing).
- The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.

| MOV <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>} |
|-----------------|-----------------|
| MOV <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>} |

is equivalent to

CPY <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the “imm8” field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:  

<table>
<thead>
<tr>
<th>sh &lt;shift&gt;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.

Operational Information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
MOV (immediate, predicated, merging)

Move signed integer immediate to vector elements (merging).

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 0 0 0 0 0 1 0 1 | size | 0 1 | Pg | 0 1 | sh | imm8 | Zd |

MOV <Zd>.<T>, <Pg>/M, #<imm>{}, <shift>}

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #<imm>{}, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MOV (scalar, predicated)

Move general-purpose register to vector elements (predicated).

Move the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of **CPY (scalar)**. This means:

- The encodings in this description are named to match the encodings of **CPY (scalar)**.
- The description of **CPY (scalar)** gives the operational pseudocode for this instruction.

![Encoding](image)

**MOV** <Zd>.<T>, <Pg>/M, <R><n|SP>

is equivalent to

**CPY** <Zd>.<T>, <Pg>/M, <R><n|SP>

and is always the preferred disassembly.

### Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<R>` Is a width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

- `<n|SP>` Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

### Operation

The description of **CPY (scalar)** gives the operational pseudocode for this instruction.

### Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a **MOVPRFX** instruction. The **MOVPRFX** instruction must conform to all of the following requirements, otherwise the behavior of the **MOVPRFX** and this instruction is **UNPREDICTABLE**.
The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

The MOVPRFX instruction must specify the same destination register as this instruction.

The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV (SIMD&FP scalar, predicated)

Move SIMD&FP scalar register to vector elements (predicated).

Move the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of CPY (SIMD&FP scalar). This means:

- The encodings in this description are named to match the encodings of CPY (SIMD&FP scalar).
- The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| MOV <Zd>.<T>, <Pg>/M, <V><n>               | MOV<Zd>.<T>,<Pg>/M,<V><n> | MOV<Zd>.<T>,<Pg>/M,<V><n> |

is equivalent to

CPY <Zd>.<T>, <Pg>/M, <V><n>

and is always the preferred disassembly.

Assembler Symbols

- **<Zd>** is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<V>** is a width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<n>** is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

Operation

The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MOV (immediate, unpredicated)

Move signed immediate to vector elements (unpredicated).

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The description of DUP (immediate) gives the operational pseudocode for this instruction.

![Encoding Table](image)

**Operation**

The description of DUP (immediate) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (scalar, unpredicated)

Move general-purpose register to vector elements (unpredicated).

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This is an alias of DUP (scalar). This means:

• The encodings in this description are named to match the encodings of DUP (scalar).
• The description of DUP (scalar) gives the operational pseudocode for this instruction.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 1 size 1 0 0 0 0 0 0 0 1 1 1 0 | Rn | Zd

MOV <Zd>, <T>, <R><n|SP>

is equivalent to

DUP <Zd>, <T>, <R><n|SP>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>W</td>
</tr>
<tr>
<td>x0</td>
<td>W</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
</tbody>
</table>

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the “Rn” field.

Operation

The description of DUP (scalar) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**MOV (SIMD&FP scalar, unpredicated)**

Move indexed element or SIMD&FP scalar to vector (unpredicated).

Unconditionally broadcast the SIMD&FP scalar into each element of the destination vector. This instruction is unpredicated.

This is an alias of **DUP (indexed)**. This means:

- The encodings in this description are named to match the encodings of **DUP (indexed)**.
- The description of **DUP (indexed)** gives the operational pseudocode for this instruction.

```
MOV <Zd>.<T>, <Zn>.<T>[<imm>]
```

is equivalent to

```
DUP <Zd>.<T>, <Zn>.<T>[<imm>]
```

and is the preferred disassembly when \text{BitCount}(imm2:tsz) > 1.

```
MOV <Zd>.<T>, <V><n>
```

is equivalent to

```
DUP <Zd>.<T>, <Zn>.<T>[0]
```

and is the preferred disassembly when \text{BitCount}(imm2:tsz) == 1.

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in "tsz":
  
<table>
<thead>
<tr>
<th>tsz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
<tr>
<td>10000</td>
<td>0</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
- `<imm>` Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".
- `<V>` Is a width specifier, encoded in “tsz”:
  
<table>
<thead>
<tr>
<th>tsz</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxx10</td>
<td>H</td>
</tr>
<tr>
<td>xx100</td>
<td>S</td>
</tr>
<tr>
<td>x1000</td>
<td>D</td>
</tr>
<tr>
<td>10000</td>
<td>0</td>
</tr>
</tbody>
</table>

- `<n>` Is the number [0-31] of the source SIMD&FP register, encoded in the “Zn” field.

**Operation**

The description of **DUP (indexed)** gives the operational pseudocode for this instruction.
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (bitmask immediate)

Move logical bitmask immediate to vector (unpredicated).

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This is an alias of DUPM. This means:

- The encodings in this description are named to match the encodings of DUPM.
- The description of DUPM gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxxxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

MOV <Zd>.<T>, #<const>

is equivalent to

DUPM <Zd>.<T>, #<const>

and is the preferred disassembly when SVEMoveMaskPreferred(imm13).

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the “imm13” field.

Operation

The description of DUPM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (predicate, unpredicated)

Move predicate (unpredicated).

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Does not set the condition flags.

This is an alias of **ORR, ORRS (predicates)**. This means:

- The encodings in this description are named to match the encodings of **ORR, ORRS (predicates)**.
- The description of **ORR, ORRS (predicates)** gives the operational pseudocode for this instruction.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | Pm | 0 | 1 | Pg | 0 | Pn | 0 | Pd |
```

MOV \(<P_d>.B, <P_n>.B\)

is equivalent to

**ORR** \(<P_d>.B, <P_n>/Z, <P_n>.B, <P_n>.B\)

and is the preferred disassembly when \(S == '0' \&\& Pn == Pm \&\& Pm == Pg\).

**Assembler Symbols**

- \(<P_d>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- \(<P_n>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of **ORR, ORRS (predicates)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MOV (vector, unpredicated)**

Move vector register (unpredicated).

Move vector register. This instruction is unpredicated.

This is an alias of ORR (vectors, unpredicated). This means:

- The encodings in this description are named to match the encodings of ORR (vectors, unpredicated).
- The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | 0 1 1 | Zm 0 0 1 1 0 0 | Zn Zd
```

MOV `<Zd>` .D, `<Zn>` .D

is equivalent to

ORR `<Zd>` .D, `<Zn>` .D, `<Zn>` .D

and is the preferred disassembly when `Zn == Zm`.

**Assembler Symbols**

`<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.

`<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.

**Operation**

The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
MOV (predicate, predicated, merging)

Move predicates (merging).

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register remain unmodified. Does not set the condition flags.

This is an alias of SEL (predicates). This means:

- The encodings in this description are named to match the encodings of SEL (predicates).
- The description of SEL (predicates) gives the operational pseudocode for this instruction.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pm 0 1 0 0 1 0 1 0 0 0</td>
</tr>
</tbody>
</table>

MOV <Pd>.B, <Pg>/M, <Pn>.B

is equivalent to


and is the preferred disassembly when Pd == Pm.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of SEL (predicates) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MOV (vector, predicated)

Move vector elements (predicated).

Move elements from the source vector to the corresponding elements of the destination vector.Inactive elements in the destination vector register remain unmodified.

This is an alias of SEL (vectors). This means:

- The encodings in this description are named to match the encodings of SEL (vectors).
- The description of SEL (vectors) gives the operational pseudocode for this instruction.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| MOV <Zd>.<T>, <Pg>/M, <Zn>.<T> |
```

is equivalent to

```
SEL <Zd>.<T>, <Pg>, <Zn>.<T>, <Zd>.<T>
```

and is the preferred disassembly when Zd == Zm.

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.

**Operation**

The description of SEL (vectors) gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
MOVPRFX (predicated)

Move prefix (predicated).

The predicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically. Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that for performance reasons software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.

Although the operation of the instruction is defined as a simple predicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same predicate register, and have the same maximum element size (ignoring a fixed 64-bit “wide vector” operand), and the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

```
MOVPRFX <Zd>.<T>, <Pg>/<ZM>, <Zn>.<T>
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean merging = (M == '1');
```

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<ZM>` Is the predication qualifier, encoded in "M”:

<table>
<thead>
<tr>
<th>M</th>
<th>&lt;ZM&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) dest = Z[d];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element;
    elsif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros();

Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
MOVPRFX (unpredicated)

Move prefix (unpredicated).

The unpredicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.

Although the operation of the instruction is defined as a simple unpredicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
bits(VL) result = Z[n];
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
**MOV$S$ (predicated)**

Move predicates (zeroing), setting the condition flags.

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of **AND, ANDS (predicates)**. This means:

- The encodings in this description are named to match the encodings of **AND, ANDS (predicates)**.
- The description of **AND, ANDS (predicates)** gives the operational pseudocode for this instruction.

![Predicate Register Encoding](image)

```
MOV$S$ <Pd>.B, <Pg>/Z, <Pn>.B
```

is equivalent to

```
```

and is the preferred disassembly when $S == '1' \&\& Pn == Pm.$

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of **AND, ANDS (predicates)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
MOV (unpredicated)

Move predicate (unpredicated), setting the condition flags.

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of ORR, ORRS (predicates). This means:

- The encodings in this description are named to match the encodings of ORR, ORRS (predicates).
- The description of ORR, ORRS (predicates) gives the operational pseudocode for this instruction.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | Pm | 0  | 1  | Pg | 0  | Pn | 0  | Pd |
```

MOV <Pd>.B, <Pn>.B

is equivalent to


and is the preferred disassembly when S == '1' && Pn == Pm && Pm == Pg.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of ORR, ORRS (predicates) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Multiply-subtract vectors (predicated), writing multiplicand \([Zdn = Z_a - Zdn * Z_m]\).

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>Pg</th>
<th>Za</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0</td>
<td>1 1</td>
<td>T</td>
<td>/M</td>
<td>0</td>
</tr>
</tbody>
</table>

MSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = TRUE;

Assembler Symbols

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.
- **<Za>** Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

CheckSVEEnabled():
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[a];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if ElemP[mask, e, esize] == '1' then
    integer product = element1 * element2;
    if sub_op then
      Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
      Elem[result, e, esize] = Elem[operand3, e, esize] + product;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**MUL (vectors, predicated)**

Multiply vectors (predicated).

Multiply active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

MUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
   integer element1 = UInt(Elem[operand1, e, esize]);
   integer element2 = UInt(Elem[operand2, e, esize]);
   if Elem[mask, e, esize] == '1' then
      integer product = element1 * element2;
      Elem[result, e, esize] = product<esize-1:0>;
   else
      Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MUL (immediate)

Multiply by immediate (unpredicated).

Multiply by an immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

```
0 1 0 0 0 0 1 1 0 | size 1 1 0 0 0 1 1 0
0 0 1 1 0 0 0 0 1 | imm8 6 5 4 3 2 1 0
Zdn
```

MUL <Zdn>.<T>, <Zdn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = SInt(imm8);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
integer element1 = SInt(Elem[operand1, e, esize]);
Elem[result, e, esize] = (element1 * imm)<esize-1:0>;
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
MUL (vectors, unpredicated)

Multiply vectors (unpredicated).

Multiply all elements of the first source vector by corresponding elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 0 1 0 0 | size | 1 | 0 1 1 0 0 0 | Zn | Zd |

MUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

- <Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
- <T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- <Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
- <Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  integer product = element1 * element2;
  Elem[result, e, esize] = product<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**MUL (indexed)**

Multiply (indexed).

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.

It has encodings from 3 classes: **16-bit**, **32-bit** and **64-bit**

### 16-bit

| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | i3h | i3l | Zm | 1 | 1 | 1 | 1 | 1 | 0 | Zn | Zd |

if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

### 32-bit

| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | i2 | Zm | 1 | 1 | 1 | 1 | 1 | 0 | Zn | Zd |

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

### 64-bit

| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | i1 | Zm | 1 | 1 | 1 | 1 | 1 | 0 | Zn | Zd |

if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**NAND, NANDS**

Bitwise NAND predicates.

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: **Not setting the condition flags** and **Setting the condition flags**

### Not setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Pm 0 1 | Pg 1 | Pn 1 | Pd |

\[
\text{NAND } \langle Pd \rangle.B, \langle Pg \rangle/Z, \langle Pn \rangle.B, \langle Pm \rangle.B
\]

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

### Setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Pm 0 1 | Pg 1 | Pn 1 | Pd |

\[
\text{NANDS } \langle Pd \rangle.B, \langle Pg \rangle/Z, \langle Pn \rangle.B, \langle Pm \rangle.B
\]

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

### Assembler Symbols

- **<Pd>**  
  Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<Pg>**  
  Is the name of the governing scalable predicate register, encoded in the "Pg" field.
- **<Pn>**  
  Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- **<Pm>**  
  Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 AND element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
Bitwise inverted select.

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The inverted result is placed destructively in the destination and first source vector. This instruction is unpredicated.

### Assembler Symbols

- `<Zdn>`: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Zk>`: Is the name of the third source scalable vector register, encoded in the "Zk" field.

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

```plaintext

if !HaveSVE2() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Z[dn] = NOT((operand1 AND operand3) OR (operand2 AND NOT(operand3)));```

"If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

```plaintext

if !HaveSVE2() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);

Z[dn] = NOT((operand1 AND operand3) OR (operand2 AND NOT(operand3)));```
NEG

Negate (predicated).

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

NEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    integer element = SInt(Elem[operand, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        element = -element;
    Elem[result, e, esize] = element<esize-1:0>;
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Detect no matching elements, setting the condition flags.

This instruction compares each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. Where the first source element detects no matching characters in the second segment it places true in the corresponding element of the destination predicate, otherwise false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 size 1 Zm 1 0 0 Pg Zn 1 0 Pd
```

NMATCH <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
if size == '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Pd);
integer n = UInt(Zn);
integer m = UInt(Zm);

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(PL) result;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer segmentbase = e - (e MOD eltspersegment);
    ElemP[result, e, esize] = '1';
    bits(esize) element1 = Elem[operand1, e, esize];
    for i = segmentbase to segmentbase + eltspersegment - 1
      bits(esize) element2 = Elem[operand2, i, esize];
      if element1 == element2 then
        ElemP[result, e, esize] = '0';
      else
        ElemP[result, e, esize] = '0';
  endfor
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
NOR, NORS

Bitwise NOR predicates.

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 1 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 1 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  ◦ The values of the NZCV flags.
NOT (predicate)

Bitwise invert predicate.

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of \texttt{EOR, EORS (predicates)}. This means:

- The encodings in this description are named to match the encodings of \texttt{EOR, EORS (predicates)}.
- The description of \texttt{EOR, EORS (predicates)} gives the operational pseudocode for this instruction.

\begin{verbatim}
0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1
\end{verbatim}

\texttt{NOT <Pd>.B, <Pg>/Z, <Pn>.B}

is equivalent to

\texttt{EOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B}

and is the preferred disassembly when \texttt{Pm == Pg}.

**Assembler Symbols**

\texttt{<Pd>} Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\texttt{<Pg>} Is the name of the governing scalable predicate register, encoded in the "Pg" field.

\texttt{<Pn>} Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of \texttt{EOR, EORS (predicates)} gives the operational pseudocode for this instruction.

**Operational information**

If \texttt{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
NOT (vector)

Bitwise invert vector (predicated).

Bitwise invert each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

NOT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = NOT element;
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
NOTS

Bitwise invert predicate, setting the condition flags.

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of **EOR, EORS (predicates)**. This means:

- The encodings in this description are named to match the encodings of **EOR, EORS (predicates)**.
- The description of **EOR, EORS (predicates)** gives the operational pseudocode for this instruction.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**NOTS** `<Pd>.B, <Pg>/Z, <Pn>.B` is equivalent to

**EORS** `<Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B` and is the preferred disassembly when `Pm == Pg`.

**Assembler Symbols**

- `<Pd>` is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<Pg>` is the name of the governing scalable predicate register, encoded in the "Pg" field.
- `<Pn>` is the name of the first source scalable predicate register, encoded in the "Pn" field.

**Operation**

The description of **EOR, EORS (predicates)** gives the operational pseudocode for this instruction.

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
ORN (immediate)

Bitwise inclusive OR with inverted immediate (unpredicated).

Bitwise inclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ORR (immediate) gives the operational pseudocode for this instruction.

| Zdn |imm13<12>|imm13<5:0>|<T>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxx</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>xxxxxx</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

ORN <Zdn>.<T>, <Zdn>.<T>, #<const>

is equivalent to

ORN <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>“:

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ORN, ORNS (predicates)

Bitwise inclusive OR inverted predicate.

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 0 0 1 0 1 1 0 0 0 0 | Pm | 0 | 1 | Pg | 0 | Pn | 1 | Pd |


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 0 0 1 0 1 1 0 0 0 0 | Pm | 0 | 1 | Pg | 0 | Pn | 1 | Pd |


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

\begin{verbatim}
CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(PL) mask = \texttt{P}[g];
bits(PL) operand1 = \texttt{P}[n];
bits(PL) operand2 = \texttt{P}[m];
bits(PL) result;
for e = 0 to elements-1
    bit element1 = \texttt{ElemP}[operand1, e, esize];
    bit element2 = \texttt{ElemP}[operand2, e, esize];
    if \texttt{ElemP}[mask, e, esize] == '1'
        \texttt{ElemP}[result, e, esize] = element1 OR (NOT element2);
    else
        \texttt{ElemP}[result, e, esize] = '0';
if setflags then
    PSTATE.<N,Z,C,V> = \texttt{PredTest}(mask, result, esize);
P[d] = result;
\end{verbatim}

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  - The values of the NZCV flags.
ORR, ORRS (predicates)

Bitwise inclusive OR predicate.

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is used by the aliases MOV (unpredicated), and MOV (predicate, unpredicated).

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

Not setting the condition flags

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  |
```


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  |
```


if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (unpredicated)</td>
<td>( S == '1' ) &amp;&amp; ( Pn == Pm &amp;&amp; Pm == Pg )</td>
</tr>
<tr>
<td>MOV (predicate, unpredicated)</td>
<td>( S == '0' ) &amp;&amp; ( Pn == Pm &amp;&amp; Pm == Pg )</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits PL mask = P[g];
bits PL operand1 = P[n];
bits PL operand2 = P[m];
bits PL result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
**ORR (vectors, predicated)**

Bitwise inclusive OR vectors (predicated).

Bitwise inclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**ORR** `<Zdn>`<T>, `<Pg>/M`, `<Zm>`<T>, `<Zm>`<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 OR element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
    Z[dn] = result;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ORR (immediate)

Bitwise inclusive OR with immediate (unpredicated).

Bitwise inclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction ORN (immediate).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 1 0 0 0 0 0 imm13 Zdn
```

ORR <Zdn>.<T>, <Zdn>.<T>, #<const>

```
if !HaveSVE() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

<table>
<thead>
<tr>
<th>imm13&lt;12&gt;</th>
<th>imm13&lt;5:0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xxxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxx</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>110xxx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1110xx</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>11110x</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>111110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>xxxxxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV 64;
bits(VL) operand = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 OR imm;
Z[dn] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**ORR (vectors, unpredicated)**

Bitwise inclusive OR vectors (unpredicated).

Bitwise inclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the first in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction is used by the alias **MOV (vector, unpredicated)**.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembler Symbols**

- `<Zd>` is the name of the destination scalable vector register, encoded in the “Zd” field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the “Zm” field.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOV (vector, unpredicated)</strong></td>
<td>Zn == Zm</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
Z[d] = operand1 OR operand2;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**ORV**

Bitwise inclusive OR reduction to scalar.

Bitwise inclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 | size | 0 1 1 0 0 0 0 0 1 | Pg | Zn | Vd
```

ORV <V><d>, <Pg>, <Zn><T>

```c
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

**Assembler Symbols**

- **<V>** Is a width specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<d>** Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.

- **<T>** Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

- `CheckSVEEnabled();`
- `integer elements = VL DIV esize;`
- `bits(PL) mask = P[g];`
- `bits(VL) operand = Z[n];`
- `bits(esize) result = Zeros(esize);`
- `for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    result = result OR Elem[operand, e, esize];`
- `V[d] = result;`

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
The values of the NZCV flags.
Set all predicate elements to false.

Set all elements in the destination predicate to false.

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

if !HaveSVE() then UNDEFINED;
integer d = UInt(Pd);

P[d] = Zeros(PL);

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
PFIRST

Set the first active predicate element to true.

Sets the first active element in the destination predicate to true, otherwise elements from the source predicate are passed through unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1 0 1 0 1 1 0 0 0 1 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>S Pg</td>
</tr>
</tbody>
</table>

PFIRST <Pdn>.B, <Pg>, <Pdn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer dn = UInt(Pdn);

Assembler Symbols

<Pdn> Is the name of the source and destination scalable predicate register, encoded in the "Pdn" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) result = P[dn];
integer first = -1;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' && first == -1 then
    first = e;

if first >= 0 then
  ElemP[result, first, esize] = '1';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn] = result;

PMUL

Polynomial multiply vectors (unpredicated).

Polynomial multiply over [0, 1] all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 1   | 0   | 0   | 0   | 1   | Zm  | 0   | 1   | 0   | 0   | 1   | Zn  | Zd  |
```

PMUL <Zd>.B, <Zn>.B, <Zm>.B

if !HaveSVE2() then UNDEFINED;
integer esize = 8;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = PolynomialMult(element1, element2)<esize-1:0>;
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
PMULLB

Polynomial multiply long (bottom).

Polynomial multiply over \([0, 1]\) the corresponding even-numbered elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated. ID AA64ZFR0_EL1.AES indicates whether the instruction variant with 64-bit source and 128-bit destination elements is implemented.

Assembler Symbols

\(<Zd>\)  
Is the name of the destination scalable vector register, encoded in the “Zd” field.

\(<T>\)  
Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Q</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Zn>\)  
Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<Tb>\)  
Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>D</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

\(<Zm>\)  
Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

\(\text{CheckSVEEnabled}()\);
integer elements = \(\text{VL} \div \text{esize}\);
bits(\(\text{VL}\)) operand1 = \(\text{Z}[n]\);
bits(\(\text{VL}\)) operand2 = \(\text{Z}[m]\);
bits(\(\text{VL}\)) result;

for \(e = 0\) to elements-1
  bits(esize DIV 2) element1 = \(\text{Elem}[\text{operand1}, 2^e + 0, \text{esize DIV 2}]\);
  bits(esize DIV 2) element2 = \(\text{Elem}[\text{operand2}, 2^e + 0, \text{esize DIV 2}]\);
  \(\text{Elem}[\text{result, e, esize}] = \text{PolynomialMult}(\text{element1, element2})\);

\(\text{Z}[d] = \text{result}\);
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**PMULLT**

Polynomial multiply long (top).

Polynomial multiply over [0, 1] the corresponding odd-numbered elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated. ID_AA64ZFP0_EL1.AES indicates whether the instruction variant with 64-bit source and 128-bit destination elements is implemented.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 0 0 1 0 1| size            | Zm              | 0 1 1 0 1 1     | Zn              | Zd              |

PMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' & !HaveSVE2PMULL128() then UNDEFINED;

```c
case size of
  when '00' esize = 128;
  when '01' esize = 16;
  when '10' UNDEFINED;
  when '11' esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Q</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>D</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
  bits(esize DIV 2) element1 = Elem[operand1, 2*e + 1, esize DIV 2];
  bits(esize DIV 2) element2 = Elem[operand2, 2*e + 1, esize DIV 2];
  Elem[result, e, esize] = PolynomialMult(element1, element2);
Z[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
PNEXT

Find next active predicate.

An instruction used to construct a loop which iterates over all active elements in a predicate. If all source predicate elements are false it sets the first active predicate element in the destination predicate to true. Otherwise it determines the next active predicate element following the last true source predicate element, and if one is found sets the corresponding destination predicate element to true. All other destination predicate elements are set to false. Sets the \texttt{FIRST} \( (N) \), \texttt{NONE} \( (Z) \), \texttt{!LAST} \( (C) \) condition flags based on the predicate result, and the \texttt{V} flag to zero.

\begin{verbatim}
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | size 0 1 1 0 0 1 1 1 0 0 0 1 0 | Pg 0 | Pdn
\end{verbatim}

\begin{verbatim}
PNEXT <Pdn>.<T>, <Pg>, <Pdn>.<T>
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Pdn);
\end{verbatim}

Assembler Symbols

\begin{itemize}
\item \texttt{<Pdn>} Is the name of the source and destination scalable predicate register, encoded in the "Pdn" field.
\item \texttt{<T>} Is the size specifier, encoded in "size":
\begin{verbatim}
size <T>
00 B
01 H
10 S
11 D
\end{verbatim}
\item \texttt{<Pg>} Is the name of the governing scalable predicate register, encoded in the "Pg" field.
\end{itemize}

Operation

\begin{verbatim}
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand = P[dn];
bits(PL) result;
integer next = LastActiveElement(operand, esize) + 1;
while next < elements \&\& (ElemP[mask, next, esize] == '0') do
  next = next + 1;
result = Zeros();
if next < elements then
  ElemP[result, next, esize] = '1';
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn] = result;
\end{verbatim}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFB (vector plus immediate)

Gather prefetch bytes (vector plus immediate).

Gather prefetch of bytes from the active memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive addresses are not prefetched from memory.
The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>prfop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0>

PRFB <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
text offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>0</td>
<td>prfop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1>msz<0>

PRFB <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
text offset = UInt(imm5);

Assembler Symbols

<prfop>    Is the prefetch operation specifier, encoded in "prfop":

Page 2170
<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

< Pg >  
Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

< Zn >  
Is the name of the base scalable vector register, encoded in the "Zn" field.

< imm >  
Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(64) addr;
base = Z[n];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFB (scalar plus immediate)

Contiguous prefetch bytes (immediate index).

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 0 1 0 1 1 1 imm6 0 0 0 Pg Rn 0 prfop

msz<1>msz<0>

PRFB <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = SInt(imm6);

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x  #uimm4</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) addr;

if n == 31 then
  base = SP[];
else
  base = X[n];

addr = base + ((offset * elements) << scale);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Hint_Prefetch(addr, pref_hint, level, stream);
  addr = addr + (1 << scale);
```

PRFB (scalar plus scalar)

Contiguous prefetch bytes (scalar index).

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

If $\text{HaveSVE}()$ then UNDEFINED;
if $Rm = '11111'$ then UNDEFINED;
integer esize = 8;
integer $g = \text{UInt}(Pg)$;
integer $n = \text{UInt}(Rn)$;
integer $m = \text{UInt}(Rm)$;
integer level = $\text{UInt}(\text{prfop}<2:1>)$;
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;

Assembler Symbols

$\text{prfop}$ is the prefetch operation specifier, encoded in ”prfop”:

<table>
<thead>
<tr>
<th>$\text{prfop}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

$\text{Pg}$ is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

$\text{Xn|SP}$ is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

$\text{Xm}$ is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits[PL] mask = P[g];
bits(64) base;
bits(64) offset = X[m];
bits(64) addr;

if n == 31 then
    base = SP[];
else
    base = X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = base + (UInt(offset) << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
        offset = offset + 1;
```
**PRFB (scalar plus vector)**

Gather prefetch bytes (scalar plus vector).

Gather prefetch of bytes from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive addresses are not prefetched from memory.

The `<prfop>` symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset and 64-bit scaled offset

### 32-bit scaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0 0 0</td>
</tr>
</tbody>
</table>
```

msz<1>msz<0>

If !HaveSVE() then UNDEFINED;

integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);

boolean stream = (prfop<0> == '1');

pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;

integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;

### 32-bit unpacked scaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0</td>
</tr>
</tbody>
</table>
```

msz<1>msz<0>

If !HaveSVE() then UNDEFINED;

integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);

boolean stream = (prfop<0> == '1');

pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;

integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;

### 64-bit scaled offset

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0 0 0 1</td>
</tr>
</tbody>
</table>
```

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');

pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

**Assembler Symbols**

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits([PL]) mask = P[g];
bits(64) base;
bits(64) addr;
bits(VL) offset;
if n == 31 then
  base = SP[);
else
  base = X[n];
offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
PRFD (vector plus immediate)

Gather prefetch doublewords (vector plus immediate).

Gather prefetch of doublewords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive addresses are not prefetched from memory. The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | imm5| 1  | 1  | 1  | Pg | Zn | 0  | prfop |

```
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

64-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | imm5| 1  | 1  | 1  | Pg | Zn | 0  | prfop |

```
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:
<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(64) addr;
base = Z[n];
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFD (scalar plus immediate)

Contiguous prefetch doublewords (immediate index).

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

PRFD <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>prfop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) addr;

if n == 31 then
    base = SP[];
else
    base = X[n];

addr = base + ((offset * elements) << scale);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Hint_Prefetch(addr, pref_hint, level, stream);
    addr = addr + (1 << scale);
PRFD (scalar plus scalar)

Contiguous prefetch doublewords (scalar index).

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| 1 0 0 0 0 1 0 1 1 0 0  0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 0 1 1 0 0 0 | Rm | Pg | Rn | prfop |

msz<1>msz<0>

PRFD <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;

Assembler Symbols

<prfop>  Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>&lt;prfop&gt;</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits PL mask = P[g];
bits(64) base;
bits(64) offset = X[m];
bits(64) addr;

if n == 31 then
    base = SP[];
else
    base = X[n];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = base + (UInt(offset) << scale);
        Hint_PREFETCH(addr, pref_hint, level, stream);
        offset = offset + 1;
Gather prefetch doublewords (scalar plus vector).

Gather prefetch of doublewords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 8. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset and 64-bit scaled offset

### 32-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 0 0 0 0 1 0 0 0(xs)1 | Zm | 0 | 1 | 1 | Pg | Rn | 0 | prfop |

msz<1>msz<0>

\[
\text{PRFD} \langle \text{prfop} \rangle, \langle \text{Pg} \rangle, \langle \text{Xn}\mid\text{SP} \rangle, \langle \text{Zm} \rangle . \text{S}, \langle \text{mod} \rangle \#3
\]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;

### 32-bit unpacked scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 0 0 0 1 0 0 0(xs)1 | Zm | 0 | 1 | 1 | Pg | Rn | 0 | prfop |

msz<1>msz<0>

\[
\text{PRFD} \langle \text{prfop} \rangle, \langle \text{Pg} \rangle, \langle \text{Xn}\mid\text{SP} \rangle, \langle \text{Zm} \rangle . \text{D}, \langle \text{mod} \rangle \#3
\]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;

### 64-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 1 1 0 0 0 1 0 0 0 1 | Zm | 1 | 1 | 1 | Pg | Rn | 0 | prfop |

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#imm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
in
teger elements = VL DIV esize;
b\[PL\] mask = P[g];
bits(64) base;
bits(64) addr;
bits(VL) offset;
if n == 31 then
    base = SP[ ];
else
    base = X[n];
offset = Z[m];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
PRFH (vector plus immediate)

Gather prefetch halfwords (vector plus immediate).

Gather prefetch of halfwords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive addresses are not prefetched from memory. The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 2 classes: 32-bit element and 64-bit element

**32-bit element**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 0 0 0 1 0 | 0 | 1 | 0 | 0 | imm5 | 1 | 1 | 1 | Pg | Zn | 0 | prfop |
| msz<1>msz<0> |

PRFH <prfop>, <Pg>, [<Zn>].S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);

**64-bit element**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 0 0 1 0 | 0 | 1 | 0 | 0 | imm5 | 1 | 1 | 1 | Pg | Zn | 0 | prfop |
| msz<1>msz<0> |

PRFH <prfop>, <Pg>, [<Zn>].D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);

**Assembler Symbols**

<prfop> Is the prefetch operation specifier, encoded in "prfop":

---

PRF (vector plus immediate)
<table>
<thead>
<tr>
<th>prfop</th>
<th>prfop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(64) addr;
base = Z[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
      addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
      Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**PRFH (scalar plus immediate)**

Contiguous prefetch halfwords (immediate index).

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 1 0 0 0 0 1 0 1 1 imm6 0 0 1 Pg  Rn  0  prfop
```

If `HaveSVE()` then UNDEFINED;

- integer esize = 16;
- integer g = UInt(Pg);
- integer n = UInt(Rn);
- integer level = UInt(prfop<2:1>);
- boolean stream = (prfop<0> == '1');
- pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
- integer scale = 1;
- integer offset = SInt(imm6);

### Assembler Symbols

**<prfop>** Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

**<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

**<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

**<imm>** Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
Operation

`CheckSVEEnabled();`

integer elements = \texttt{VL} \texttt{DIV} esize;

bits(\texttt{PL}) mask = \texttt{P}[g];

bits(64) base;

bits(64) addr;

if n == 31 then
  base = \texttt{SP}[];
else
  base = \texttt{X}[n];

addr = base + \((\text{offset} \times \text{elements}) \ll \text{scale})

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    \texttt{Hint_Prefetch}(addr, pref_hint, level, stream);
  addr = addr + (1 \ll \text{scale});
PRFH (scalar plus scalar)

Contiguous prefetch halfwords (scalar index).

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------------------------|-------------------|
| 1 0 0 0 0 1 0 1 0 0 Rm 1 1 0 Pg Rn 0 prfop | msz<1> msz<0> |

PRFH <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) offset = X[m];
bits(64) addr;

if n == 31 then
  base = SP[ ];
else
  base = X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + (UInt(offset) << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
  offset = offset + 1;
```
PRFH (scalar plus vector)

Gather prefetch halfwords (scalar plus vector).

Gather prefetch of halfwords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 2. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset and 64-bit scaled offset

32-bit scaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | xs | 1  | Zm | 0  | 0  | 1  | Pg | Rn | 0  | prfop |
```

```
PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;
```

32-bit unpacked scaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | xs | 1  | Zm | 0  | 0  | 1  | Pg | Rn | 0  | prfop |
```

```
PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;
```

64-bit scaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | Zm | 1  | 0  | 1  | Pg | Rn | 0  | prfop |
```

msz<1>msz<0>
if ! HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 1;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) addr;
bits(VL) offset;
if n == 31 then
    base = SP[];
else
    base = X[n];
offset = Z[m];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
PRFW (vector plus immediate)

Gather prefetch words (vector plus immediate).

Gather prefetch of words from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive addresses are not prefetched from memory. The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 2 classes: 32-bit element and 64-bit element

### 32-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 0 1 0</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

**msz<1>** msz<0>

PRFW <prfop>, <Pg>, [<Zn>].S{, #<imm>}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);

### 64-bit element

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 1 0</td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

**msz<1>** msz<0>

PRFW <prfop>, <Pg>, [<Zn>].D{, #<imm>}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);

### Assembler Symbols

**<prfop>** Is the prefetch operation specifier, encoded in "prfop":

PRFW (vector plus immediate)
<prfop> <prfop>
0000 PLDL1KEEP
0001 PLDL1STRM
0010 PLDL2KEEP
0011 PLDL2STRM
0100 PLDL3KEEP
0101 PLDL3STRM
x11x #uimm4
1000 PSTL1KEEP
1001 PSTL1STRM
1010 PSTL2KEEP
1011 PSTL2STRM
1100 PSTL3KEEP
1101 PSTL3STRM
</pg>  

Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<zn>  
Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm>  
Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckSVEEnabled();
integer elements = VL / esize;
bits(PL) mask = P[g];
bits(VL) base;
bits(64) addr;
base = Z[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
PRFW (scalar plus immediate)

Contiguous prefetch words (immediate index).

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector’s in-memory size, irrespective of predication, and added to the base address.

The predicate may be used to suppress prefetches from unwanted addresses.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1 0 0 0 0 1 0 1 1 1 imm6 0 1 0 Pg Rn 0 prfop

msz<1>msz<0>

PRFW <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = SInt(imm6);

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:  

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bias(64) base;
bias(64) addr;

if n == 31 then
    base = SP[];
else
    base = X[n];

addr = base + ((offset * elements) << scale);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Hint_Prefetch(addr, pref_hint, level, stream);
    addr = addr + (1 << scale);
PRFW (scalar plus scalar)

Contiguous prefetch words (scalar index).

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|----------------|----------------|
| Rm | Pg | Rn | prfop |
```

msz<1>msz<0>

PRFW <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```c
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
```

Assembler Symbols

- `<prfop>` is the prefetch operation specifier, encoded in “prfop”:

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

- `<Xm>` is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P) mask = P[g];
bits(64) base;
bits(64) offset = X[m];
bits(64) addr;

if n == 31 then
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = base + (UInt(offset) << scale);
    Hint_PREFETCH(addr, pref_hint, level, stream);
    offset = offset + 1;
PRFW (scalar plus vector)

Gather prefetch words (scalar plus vector).

Gather prefetch of words from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 4. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

It has encodings from 3 classes: 32-bit scaled offset, 32-bit unpacked scaled offset and 64-bit scaled offset

32-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 0 0 0 0 1 0 0 0 xs 1 | Zm | 0 | 1 | 0 | Pg | Rn | 0 | prfop |

msz<1>msz<0>

PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;

32-bit unpacked scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 0 0 1 0 0 0 xs 1 | Zm | 0 | 1 | 0 | Pg | Rn | 0 | prfop |

msz<1>msz<0>

PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;

64-bit scaled offset

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 1 1 0 0 0 1 0 0 0 1 1 | Zm | 1 | 1 | 0 | Pg | Rn | 0 | prfop |

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');

pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in "prfop":

<table>
<thead>
<tr>
<th>prfop</th>
<th>&lt;prfop&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>PLDL1KEEP</td>
</tr>
<tr>
<td>0001</td>
<td>PLDL1STRM</td>
</tr>
<tr>
<td>0010</td>
<td>PLDL2KEEP</td>
</tr>
<tr>
<td>0011</td>
<td>PLDL2STRM</td>
</tr>
<tr>
<td>0100</td>
<td>PLDL3KEEP</td>
</tr>
<tr>
<td>0101</td>
<td>PLDL3STRM</td>
</tr>
<tr>
<td>x11x</td>
<td>#uimm4</td>
</tr>
<tr>
<td>1000</td>
<td>PSTL1KEEP</td>
</tr>
<tr>
<td>1001</td>
<td>PSTL1STRM</td>
</tr>
<tr>
<td>1010</td>
<td>PSTL2KEEP</td>
</tr>
<tr>
<td>1011</td>
<td>PSTL2STRM</td>
</tr>
<tr>
<td>1100</td>
<td>PSTL3KEEP</td>
</tr>
<tr>
<td>1101</td>
<td>PSTL3STRM</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(64) base;
bits(64) addr;
bits(VL) offset;
if n == 31 then
  base = SP[];
else
  base = X[n];
offset = Z[m];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
Set condition flags for predicate.

Sets the **FIRST** (N), **NONE** (Z), !**LAST** (C) condition flags based on the predicate source register, and the V flag to zero.

```
0 0 1 0 0 1 0 1 0 1 0 0 0 0 1 1  PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
```

**Assembler Symbols**

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

```
CheckSVEEnabled();
bits(P) mask = P[g];
bits(P) result = P[n];
PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
**PTRUE, PTRUES**

Initialise predicate from named constraint.

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Optionally sets the first (N), none (Z), !last (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

### Not setting the condition flags

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 size | 0 1 1 0 0 0 1 1 1 0 0 0 pattern | 0 | Pd
```

```
PTRUE <Pd>.<T>{, <pattern>}
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = FALSE;
bits(5) pat = pattern;

### Setting the condition flags

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 size | 0 1 1 0 0 1 1 1 0 0 0 pattern | 0 | Pd
```

```
PTRUES <Pd>.<T>{, <pattern>}
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = TRUE;
bits(5) pat = pattern;

### Assembler Symbols

- **<Pd>** Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- **<T>** Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

- **<pattern>** Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

---

---
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uiimm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#uiimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uiimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uiimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uiimm5</td>
</tr>
<tr>
<td>11111</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;

for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
PUNKHI, PUNKKLO

Unpack and widen half of predicate.

Unpack elements from the lowest or highest half of the source predicate and place in elements of twice their size within the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: **High half** and **Low half**

**High half**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Pn| 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Pd| 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

PUNKHI <Pd>.H, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = TRUE;

**Low half**

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Pn| 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Pd| 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

PUNKKLO <Pd>.H, <Pn>.B

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = FALSE;

**Assembler Symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) operand = P[n];
bits(PL) result;
for e = 0 to elements-1
    ElemP[result, e, esize] = ElemP[operand, if hi then e + elements else e, esize DIV 2];
P[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
RADDHNB

Rounding add narrow high part (bottom).

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 1 & \text{Zm} & 0 & 1 & 1 & 0 & 1 & 0 & \text{Zn} & \text{Zd} & \text{S} & \text{R} & \text{T}
\end{array}
\]

\text{RADDHNB} <\text{Zd}>.<<\text{T}>, <\text{Zn}>>.<<\text{ Tb}>, <\text{Zm}>>.<<\text{T}b>

if \text{ !HaveSVE2}() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer n = \text{UInt}(\text{Zn});
integer m = \text{UInt}(\text{Zm});
integer d = \text{UInt}(\text{Zd});

\textbf{Assembler Symbols}

<\text{Zd}> Is the name of the destination scalable vector register, encoded in the "\text{Zd}" field.

<\text{T} > Is the size specifier, encoded in "\text{size}":

\begin{table}[h]
\begin{tabular}{c|c}
\hline
\text{size} & \text{<\text{T}>} \\
\hline
00 & RESERVED \\
01 & B \\
10 & H \\
11 & S \\
\hline
\end{tabular}
\end{table}

<\text{Zn}> Is the name of the first source scalable vector register, encoded in the "\text{Zn}" field.

<\text{Tb}> Is the size specifier, encoded in "\text{size}":

\begin{table}[h]
\begin{tabular}{c|c}
\hline
\text{size} & \text{<\text{Tb}>} \\
\hline
00 & RESERVED \\
01 & H \\
10 & S \\
11 & D \\
\hline
\end{tabular}
\end{table}

<\text{Zm}> Is the name of the second source scalable vector register, encoded in the "\text{Zm}" field.

\textbf{Operation}

\textbf{CheckSVEEnabled}();
integer elements = \text{VL} \text{ DIV esize};
bits(\text{VL}) operand1 = \text{Z}[n];
bits(\text{VL}) operand2 = \text{Z}[m];
bits(\text{VL}) result;
integer halffsize = esize \text{ DIV} 2;
integer \text{round const} = 1 << (halffsize - 1);

for e = 0 to elements-1
  integer element1 = \text{UInt}(\text{Elem}(\text{operand1}, e, \text{esize}));
  integer element2 = \text{UInt}(\text{Elem}(\text{operand2}, e, \text{esize}));
  integer res = ((element1 + element2) + \text{round const}) \gg \text{halffsize};
  \text{Elem}[\text{result, 2*e + 0, halffsize}] = \text{res<halffsize-1:0>};
  \text{Elem}[\text{result, 2*e + 1, halffsize}] = \text{Zeros}();

\text{Z[d]} = \text{result};
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RADDHNT

Rounding add narrow high part (top).

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

RADDHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[d];
integer halfesize = esize DIV 2;
integer round_const = 1 << (halfesize - 1);

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 + element2) + round const) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
RAX1

Bitwise rotate left by 1 and exclusive OR.

Rotate each 64-bit element of the second source vector left by 1 and exclusive OR with the corresponding elements of the first source vector. The results are placed in the corresponding elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SHA3 indicates whether this instruction is implemented.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV 64;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(64) element1 = Elem[operand1, e, 64];
    bits(64) element2 = Elem[operand2, e, 64];
    Elem[result, e, 64] = element1 EOR ROL(element2, 1);
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RBIT

Reverse bits (predicated).

Reverse bits in each active element of the source vector, and place the results in the corresponding elements of the
destination vector. Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 size 1 0 0 1 1 1 1 0 0 Pg Zn Zd

RBIT <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
   bits(esize) element = Elem[operand, e, esize];
   if Elem[mask, e, esize] == '1' then
      Elem[result, e, esize] = BitReverse(element);
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
• The **MOVPRFX** instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
RDFR (unpredicated)

Read the first-fault register.

Read the first-fault register (FFR) and place in the destination predicate without predication.

```

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

S

RDFR <Pd>.B
```

```
if !HaveSVE() then UNDEFINED;
integer d = UInt(Pd);
```

**Assembler Symbols**

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

**Operation**

```
CheckSVEEnabled();
bits(PL) ffr = FFR[];
P[d] = ffr;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**RDFFR, RDFFRS (predicated)**

Return predicate of successfully loaded elements.

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Optionally sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

It has encodings from 2 classes: Not setting the condition flags and Setting the condition flags

### Not setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Pg | 0 | Pd |
|---|---|---|
| 0 0 1 0 0 1 0 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 0 0 1 1 1 | S |

**RDFFR** <Pd>.B, <Pg>/<Z>

if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = FALSE;

### Setting the condition flags

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | Pg | 0 | Pd |
|---|---|---|
| 0 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 1 1 1 1 0 0 0 0 1 1 1 | S |

**RDFFRS** <Pd>.B, <Pg>/<Z>

if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = TRUE;

**Assembler Symbols**

**<Pd>**

Is the name of the destination scalable predicate register, encoded in the "Pd" field.

**<Pg>**

Is the name of the governing scalable predicate register, encoded in the "Pg" field.

### Operation

**CheckSVEEnabled();**

bits(PL) mask = P[g];
bits(PL) ffr = FFR[];
bits(PL) result = ffr AND mask;

if setflags then
  PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d] = result;

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate
  register contains the same value for each execution.
- The values of the NZCV flags.
**RDVL**

Read multiple of vector register size to scalar register.

Multiply the current vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | imm6 | Rd |

RDVL `<Xd>`, #<imm>

if !HaveSVE() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);

**Assembler Symbols**

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<imm> Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer len = imm * (VL DIV 8);
Xd[d] = len<63:0>;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
REV (predicate)

Reverse all elements in a predicate.

Reverse the order of all elements in the source predicate and place in the destination predicate. This instruction is unpredicated.

```
REV <Pd>.<T>, <Pn>.<T>
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer d = UInt(Pd);

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
bits(PL) operand = P[n];
bits(PL) result = Reverse(operand, esize DIV 8);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
REV (vector)

Reverse all elements in a vector (unpredicated).

Reverse the order of all elements in the source vector and place in the destination vector. This instruction is unpredicated.

```
REV <Zd>.<T>, <Zn>.<T>
```

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
bits(VL) operand = Z[n];
bits(VL) result = Reverse(operand, esize);
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**REVB, REVH, REVW**

Reverse bytes / halfwords / words within elements (predicated).

Reverse the order of 8-bit bytes, 16-bit halfwords or 32-bit words within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: **Byte**, **Halfword** and **Word**

### Byte

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 1 0 0 1 0 0 | Pg | Zn | Zd |

REVB <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 8;

### Halfword

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 1 0 1 1 0 0 | Pg | Zn | Zd |

REVH <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 16;

### Word

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 1 | size | 1 0 0 1 1 0 1 0 0 | Pg | Zn | Zd |

REVW <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer swsize = 32;

### Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` For the byte variant: is the size specifier, encoded in "size".
size <T>
| 00 | RESERVED |
| 01 | H        |
| 10 | S        |
| 11 | D        |

For the halfword variant: is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt; &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Pg Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Zn Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = Reverse(element, swsize);
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
RSHRNB

Rounding shift right narrow by immediate (bottom).

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------|---------------------------------|---------------------------------|
| 0 1 0 0 1 0 1 0 | tszh | tszl | imm3 | 0 0 0 1 1 0 | Zn | Zd |
```

RSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsz:imm3);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>RESERVE</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>1 xx</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>RESERVE</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>1 xx</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (UInt(element) + round_const) >> shift;
  Elem[result, 2*e + 0, esize] = res<esize-1:0>;
  Elem[result, 2*e + 1, esize] = Zeros();
Z[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Rounding shift right narrow by immediate (top).

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

RSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsz:imm3);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (UInt(element) + round_const) >> shift;
  Elem[result, 2*e + 1, esize] = res<esize-1:0>;
Z[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RSUBHNB

Rounding subtract narrow high part (bottom).

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered half-width destination elements to zero. This instruction is unpredicated.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Zm | Zn | Zd | S | R | T |

RSUBHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer halfesize = esize DIV 2;
integer round_const = 1 << (halfesize - 1);

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 - element2) + round_const) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros();
Z[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
RSUBHNT

Rounding subtract narrow high part (top).

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

```
0 1 0 0 1 0 1 | size 1 | Zm   | 0 1 1 1 1 1 1 | Zn   | Zd
```

RSUBHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>  Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[d];
integer halfesize = esize DIV 2;
integer round_const = 1 << (halfesize - 1);
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  integer res = ((element1 - element2) + round_const) >> halfesize;
  Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Signed absolute difference and accumulate.

Compute the absolute difference between signed integer values in elements of the second source vector and corresponding elements of the first source vector, and add the difference to the corresponding elements of the destination vector. This instruction is unpredicated.

```java
if (!HaveSVE2()) then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean unsigned = FALSE;
```

Assembler Symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE.
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SABALB

Signed absolute difference and accumulate long (bottom).

Compute the absolute difference between even-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
 0 1 0 0 0 1 0 1 | size | 0 | Zm | 1 1 0 0 0 0 0 | Zn | Zda
               U T
```

SABALB <Zda>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

- CheckSVEEnabled();
- integer elements = VL DIV esize;
- bits(VL) operand1 = Z[n];
- bits(VL) operand2 = Z[m];
- bits(VL) result = Z[da];
- for e = 0 to elements-1
  - integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
  - integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
  - bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
  - Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
- Z[da] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed absolute difference and accumulate long (top).

Compute the absolute difference between odd-numbered signed elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
0 1 0 0 0 1 0 1 | size | 0  Zm | 1 1 0 0 0 1 | Zn | Zda
0 1 0 0 0 1 0 1
```

SABALT <Zda>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SABD

Signed absolute difference (predicated).

Compute the absolute difference between signed integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Pg | Zm | Zdn |

SABD <Zdn><T>, <Pg>/M, <Zdn><T>, <Zm><T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if Elem[mask, e, esize] == '1' then
    integer absdiff = Abs(element1 - element2);
    Elem[result, e, esize] = absdiff<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed absolute difference long (bottom).

Compute the absolute difference between even-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
 0 1 0 0 0 1 0 1  |  size  |  Zm  |  0 0 1 1 0 0  |  Zn  |  Zd
```

SABDLB `<Zd>..<T>`, `<Zn>..<Tb>`, `<Zm>..<Tb>`

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = SInt(Elem(operand1, 2*e + 0, esize DIV 2));
  integer element2 = SInt(Elem(operand2, 2*e + 0, esize DIV 2));
  integer res = Abs(element1 - element2);
  Elem[result, e, esize] = res<esize-1:0>;

Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SABDLT

Signed absolute difference long (top).

Compute the absolute difference between odd-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in overlapping double-width elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 0 0 1 0 1 | size | 0 | Zm | 0 0 1 1 0 1 | Zn | Zd | S U T |

SABDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
  integer res = Abs(element1 - element2);
  Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
• The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**SADALP**

Signed add and accumulate long pairwise.

Add pairs of adjacent signed integer values and accumulate the results into the overlapping double-width elements of the destination vector.

```
   31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
   0 1 0 0 0 1 0 0 | 0 0 0 1 0 0 1 0 1 | Pg | Zn | Zda
```

SADALP `<Zda>..<T>`, `<Pg>/M`, `<Zn>..<Tb>`

```plaintext
if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer da = UInt(Zda);
```

**Assembler Symbols**

- `<Zda>` Is the name of the second source and destination scalable vector register, encoded in the "Zda" field.
- `<T>` Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand_acc = Z[da];
bits(VL) operand_src = Z[n];
bits(VL) result;

for e = 0 to elements-1
  if Elem[mask, e, esize] == '0'
    Elem[result, e, esize] = Elem[operand_acc, e, esize];
  else
    integer element1 = SInt(Elem[operand_src, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand_src, 2*e + 1, esize DIV 2]);
    bits(esize) sum = (element1 + element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[operand_acc, e, esize] + sum;

Z[da] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SADDLB

Signed add long (bottom).

Add the corresponding even-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 1</td>
</tr>
</tbody>
</table>

SADDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = FALSE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SADDLBT

Signed add long (bottom + top).

Add the even-numbered signed elements of the first source vector to the odd-numbered signed elements of the second source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | . |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
```

SADDLBT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>
```

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
  integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
  integer res = element1 + element2;
  Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SADDLT

Signed add long (top).

Add the corresponding odd-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|----------------|
| 0 1 0 0 1 0 1 | size | 0 | Zm | 0 0 0 0 1 | Zn | Zd |

SADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>  Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb>  Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SADDV

Signed add reduction to scalar.

Signed add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first sign-extended to 64 bits. Inactive elements in the source vector are treated as zero.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Vd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SADDV <Dd>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

Assembler Symbols

| <Dd>  | Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field. |
| <Pg>  | Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field. |
| <Zn>  | Is the name of the source scalable vector register, encoded in the "Zn" field. |
| <T>   | Is the size specifier, encoded in "size": |
| size  | <T> |
| 00    | B   |
| 01    | H   |
| 10    | S   |
| 11    | RESERVED |

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer sum = 0;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = SInt(Elem[operand, e, esize]);
    sum = sum + element;
V[d] = sum<63:0>;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
SADDWB

Signed add wide (bottom).

Add the even-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

```
0 1 0 0 0 1 0 1 | size 0 | Zm |
0 1 0 0 0 0 0 0 | Zn |

S U T

SADDWB <Zd>..<T>, <Zn>..<T>, <Zm>..<Tb>
```

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SADDWT

Signed add wide (top).

Add the odd-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>Size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

SADDWT <Zd>,<T>, <Zn>,<T>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == ‘00’ then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assemble Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
Subtract with carry long (bottom).

Subtract the even-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1 1</td>
</tr>
</tbody>
</table>

SBCLB <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<T> Is the size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n];
bits(VL) carries = Z[m];
bits(VL) result = Z[da];

for p = 0 to pairs-1
bits(esize) element1 = Elem[result, 2*p + 0, esize];
bits(esize) element2 = Elem[operand, 2*p + 0, esize];
bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

(res, nzcv) = AddWithCarry(element1, NOT(element2), carry_in);
carry_out = nzcv<1>;
Elem[result, 2*p + 0, esize] = res;
Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out);

Z[da] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SBCLT

Subtract with carry long (top).

Subtract the odd-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | sz | 0  | Zm | 1  | 1  | 0  | 1  | 0  | Zn | 1  | 0  | Zda|

SBCLT <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<T> Is the size specifier, encoded in “sz”:

<table>
<thead>
<tr>
<th>sz</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n];
bits(VL) carries = Z[m];
bits(VL) result = Z[da];

for p = 0 to pairs - 1
  bits(esize) element1 = Elem[result, 2*p + 0, esize];
  bits(esize) element2 = Elem[operand, 2*p + 1, esize];
  bit carry_in = Elem[carries, 2*p + 1, esize]<0>;
  (res, nzcv) = AddWithCarry(element1, NOT(element2), carry_in);
  carry_out = nzcv<1>;
  Elem[result, 2*p + 0, esize] = res;
  Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out);

Z[da] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SCVTF

Signed integer convert to floating-point (predicated).

Convert to floating-point from the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: 16-bit to half-precision , 32-bit to half-precision , 32-bit to single-precision , 32-bit to double-precision , 64-bit to half-precision , 64-bit to single-precision and 64-bit to double-precision

16-bit to half-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

int_U
```

SCVTF <Zd>.H, <Pg>/M, <Zn>.H

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR());

32-bit to half-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

int_U
```

SCVTF <Zd>.H, <Pg>/M, <Zn>.S

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR());

32-bit to single-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

int_U
```
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[ ]);
64-bit to double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1</td>
</tr>
</tbody>
</table>

SCVTF \(<\text{Zd}\), \(<\text{Pg}\>/\text{M}, \(<\text{Zn}\)\>.

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

\(<\text{Zd}\>) Is the name of the destination scalable vector register, encoded in the "Zd" field.
\(<\text{Pg}\>) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\(<\text{Zn}\>) Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

\(\text{CheckSVEEnabled}();\)
integer elements = \text{VL} \div esize;
bits(\text{PL}) mask = \text{P}[g];
bits(\text{VL}) operand = \text{Z}[n];
bits(\text{VL}) result = \text{Z}[d];

for e = 0 to elements-1
    bits(esize) element = \text{Elem}[\text{operand}, e, esize];
    if \text{ElemP}[\text{mask}, e, esize] == '1' then
        bits(d_esize) fpval = \text{FixedToFP}(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding);
        \text{Elem}[\text{result}, e, esize] = \text{ZeroExtend}(fpval);
\end{verbatim}
\text{Z}[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SDIV**

Signed divide (predicated).

Signed divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | size, Pg, Zm, Zdn |

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```plaintext
if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn =UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

SDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

```plaintext
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer quotient;
        if element2 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element1) / Real(element2));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SDIVR**

Signed reversed divide (predicated).

Signed reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Assembled Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

**CheckSVEEnabled();**

integer elements = \(V\)_L DIV esize;

bits(P\(_L\)) mask = P[g];

bits(V\(_L\)) operand1 = Z[dn];

bits(V\(_L\)) operand2 = Z[m];

bits(V\(_L\)) result;

for e = 0 to elements-1

integer element1 = \text{Int}(\text{Elem}[\text{operand1}, e, esize], unsigned);

integer element2 = \text{Int}(\text{Elem}[\text{operand2}, e, esize], unsigned);

if \text{ElemP}[\text{mask}, e, esize] == '1' then

integer quotient;

if element1 == 0 then

quotient = 0;

else

quotient = \text{RoundTowardsZero}(\text{Real}(element2) / \text{Real}(element1));

\text{Elem}[\text{result}, e, esize] = quotient<esize-1:0>;

else

\text{Elem}[\text{result}, e, esize] = \text{Elem}[\text{operand1}, e, esize];

Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The `MOVPRFX` instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SDOT (vectors)**

Signed integer dot product.

The signed integer dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |

SDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

**Assembler Symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SDOT (indexed)

Signed integer indexed dot product.

The signed integer indexed dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```
0 1 0 0 0 1 0 0  1 0 1 i2  Zm 0 0 0 0 0 0  Zn  Zda
```

```
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit

```
0 1 0 0 0 1 0 0  1 1 1 i1  Zm 0 0 0 0 0 0  Zn  Zda
```

```
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the 32-bit variant: is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the 64-bit variant: is the immediate index of a quadtuplet of four 16-bit elements within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.
Operation

CheckSVEEnabled();
integer elements = \text{VL} \text{ DIV} \text{ esize};
integer eltspersegment = 128 \text{ DIV} \text{ esize};
bits(\text{VL}) operand1 = \text{Z}[n];
bits(\text{VL}) operand2 = \text{Z}[m];
bits(\text{VL}) operand3 = \text{Z}[da];
bits(\text{VL}) result;

\begin{verbatim}
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  bits(esize) res = \text{Elem}[operand3, e, esize];
  for i = 0 to 3
    integer element1 = \text{SInt}(\text{Elem}[operand1, 4 \ast e + i, esize DIV 4]);
    integer element2 = \text{SInt}(\text{Elem}[operand2, 4 \ast s + i, esize DIV 4]);
    res = res + element1 \ast element2;
  Elem[result, e, esize] = res;
\end{verbatim}

Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SEL (predicates)

Conditionally select elements from two predicates.

Read active elements from the first source predicate and inactive elements from the second source predicate and place in the corresponding elements of the destination predicate. Does not set the condition flags.

This instruction is used by the alias MOV (predicate, predicated, merging).

SEL <Pd>.B, <Pg>, <Pn>.B, <Pm>.B

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (predicate, predicated, merging)</td>
<td>Pd == Pm</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for e = 0 to elements-1
  bit element1 = ElemP[operand1, e, esize];
  bit element2 = ElemP[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    ElemP[result, e, esize] = element1;
  else
    ElemP[result, e, esize] = element2;

P[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
SEL (vectors)

Conditionally select elements from two vectors.

Read active elements from the first source vector and inactive elements from the second source vector and place in the corresponding elements of the destination vector.

This instruction is used by the alias MOV (vector, predicated).

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 1 size 1</td>
</tr>
</tbody>
</table>

SEL <Zd>.<T>, <Pg>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector, predicated)</td>
<td>Zd == Zm</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1;
    else
        Elem[result, e, esize] = element2;
Z[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SETFFR

Initialise the first-fault register to all true.

Initialise the first-fault register (FFR) to all true prior to a sequence of first-fault or non-fault loads. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 | 0 0 1 0 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0
```

SETFFR

if !HaveSVE() then UNDEFINED;

Operation

CheckSVEEnabled();

`FFR[] = Ones(PL);`

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SHADD**

Signed halving addition.

Add active signed elements of the first source vector to corresponding signed elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>S</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
```

SHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = element1 + element2;
        Elem[result, e, esize] = res<esize:1>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SHRNBNB**

Shift right narrow by immediate (bottom).

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 1 0 0 0 1 0 1 0 tszh 1 tszl imm3 0 0 0 1 0 0 | Zn | Zd |

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>1xx</td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>1xx</td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = res<esize-1:0>;
    Elem[result, 2*e + 1, esize] = Zeros();

Z[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SHRNT**

Shift right narrow by immediate (top).

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
0 1 0 0 0 1 0 1 0 tszh | tszl | imm3 | 0 0 0 1 0 1 | Zn | Zd
```

**SHRNT** `<Zd>.<T>, <Zn>.<Tb>`, `#<const>`

```plaintext
if !HaveSVE2() then UNDEFINED;

integer tsize = tszh:tszl;

case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;

integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsz:imm3);
```

**Assembler Symbols**

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in “tszh:tszl”:
  
<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` is the size specifier, encoded in “tszh:tszl”:
  
<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<const>` is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = UInt(element) >> shift;
  Elem[result, 2*e + 1, esize] = res<esize-1:0>;

Z[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SHSUB

Signed halving subtract.

Subtract active signed elements of the second source vector from corresponding signed elements of the first source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|------------------|------------------|
| size             | Pg               | Zm               | Zdn               |

SHSUB <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>,<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = element1 - element2;
        Elem[result, e, esize] = res<esize:1>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SHSUBR

Signed halving subtract reversed vectors.

Subtract active signed elements of the first source vector from corresponding signed elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  |

R  S  U

SHSUBR <Zdn>.<T>, <Pg>./M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, e, esize]);
  if ElemP[mask, e, esize] == '1' then
    integer res = element2 - element1;
    Elem[result, e, esize] = res<esize:1>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

1. The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
2. The MOVPRFX instruction must specify the same destination register as this instruction.
3. The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SLI**

Shift left and insert (immediate).

Shift each source vector element left by an immediate value, and insert the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>Shift Immediate</th>
<th>Immediate Shift Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zn</td>
<td>,Zd</td>
</tr>
</tbody>
</table>

### Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “tszh:tszl”:
  - `tszh`<br>`tszl`<br>`<T>`
  - `00`<br>`00`<br>RESERVED
  - `00`<br>`01`<br>B
  - `00`<br>`1x`<br>H
  - `01`<br>`xx`<br>S
  - `1x`<br>`xx`<br>D
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<const>` Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

### Operation

- `CheckSVEEnabled();`
- `integer elements = VL DIV esize;`
- `bits(VL) operand = Z[n];`
- `bits(VL) result = Z[d];`

```plaintext```
for e = 0 to elements-1
    bits(esize) element1 = Elem[result, e, esize];
    bits(esize) element2 = Elem[operand, e, esize];
    bits(esize) mask = LSL(Ones(esize), shift);
    bits(esize) shiftedval = LSL(element2, shift);
    Elem[result, e, esize] = (element1 AND (NOT mask)) OR shiftedval;
```
- `Z[d] = result;`

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SM4E

SM4 encryption and decryption.

The SM4E instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, together with four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each block of data is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SM4 indicates whether this instruction is implemented.

```
0 1 0 0 0 1 0 1 0 0 1 0 0 1 1 1 1 0 0 0 | Zm    | Zdn
size<1>size<0>
```

SM4E <Zdn>.S, <Zdn>.S, <Zm>.S

if !HaveSVE2SM4() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assemble Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for s = 0 to segments-1
  bits(128) key = Elem[operand2, s, 128];
  bits(32) intval;
  bits(8) sboxout;
  bits(128) roundresult = Elem[operand1, s, 128];
  bits(32) roundkey;
  for index = 0 to 3
    roundkey = Elem[key, index, 32];
    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;
    for i = 0 to 3
      Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;
  Elem[result, s, 128] = roundresult;
Z[dn] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.

The response of this instruction to asynchronous exceptions does not vary based on:
• The values of the data supplied in any of its registers.
• The values of the NZCV flags.
SM4EKEY

SM4 key updates.

The SM4EKEY instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, along with four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SM4 indicates whether this instruction is implemented.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | Zn | 1 | Zm | 0 | 0 | Zd | size<1>size<0> |

SM4EKEY <Zd>.S, <Zm>.S, <Zn>.S

if !HaveSVE2SM4() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for s = 0 to segments-1
    bits(128) source = Elem[operand2, s, 128];
    bits(32) intval;
    bits(8) sboxout;
    bits(32) const;
    bits(128) roundresult = Elem[operand1, s, 128];
    for index = 0 to 3
        const = Elem[source, index, 32];
        intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
        for i = 0 to 3
            Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);
        intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
        intval = intval EOR roundresult<31:0>;
        roundresult<31:0> = roundresult<63:32>;
        roundresult<63:32> = roundresult<95:64>;
        roundresult<95:64> = roundresult<127:96>;
        roundresult<127:96> = intval;
    Elem[result, s, 128] = roundresult;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
SMAX (vectors)

Signed maximum vectors (predicated).

Determine the signed maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|----------------|----------------|
| 0 0 0 0 0 1 0 0 | size | 0 0 1 0 0 0 0 |
| Pg              | Zm   | Zdn            |

SMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[esize-1:0] == '1'
        integer maximum = Max(element1, element2);
        Elem[result, e, esize] = maximum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMAX (immediate)

Signed maximum with immediate (unpredicated).

Determine the signed maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | size | 1 0 1 | 0 0 | 0 1 1 0 | imm8 | Zdn
```

SMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>

```java
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMAXP

Signed maximum pairwise.

Compute the maximum value of each pair of adjacent signed integer elements within each source vector, and
interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first
source vector.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

SMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = SInt(Elem[operand1, e + 0, esize]);
      element2 = SInt(Elem[operand1, e + 1, esize]);
    else
      element1 = SInt(Elem[operand2, e - 1, esize]);
      element2 = SInt(Elem[operand2, e + 0, esize]);
    integer res = Max(element1, element2);
    Elem[result, e, esize] = res<esize-1:0>;
  Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMAXV

Signed maximum reduction to scalar.

Signed maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SMAXV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;

Assembler Symbols

<V> Is a width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer maximum = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1
    if ElemP(mask, e, esize) == '1' then
        integer element = Int(Elem(operand, e, esize), unsigned);
        maximum = Max(maximum, element);

V[d] = maximum<esize-1:0>;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  ◦ The values of the NZCV flags.
SMIN (vectors)

Signed minimum vectors (predicated).

Determine the signed minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th></th>
<th></th>
<th></th>
<th>gear</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

SMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

< Pg > Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

< Zm > Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer minimum = Min(element1, element2);
        Elem[result, e, esize] = minimum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMIN (immediate)

Signed minimum with immediate (unpredicated).

Determine the signed minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 0 1 0 1 1 0 imm8 Zdn
```

SMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the “imm8” field.

Operation

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
integer element1 = Int(Elem[operand1, e, esize], unsigned);
Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SMINP**

Signed minimum pairwise.

Compute the minimum value of each pair of adjacent signed integer elements within each source vector, and interleaver the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

```
0 1 0 0 1 0 0     size 0 1 0 1 1 0 1 0     Pg     Zm     Zdn
```

**SMINP** `<Zdn>..<T>`, `<Pg>/M`, `<Zdm>..<T>`, `<Zm>..<T>`

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  - size  <T>
    - 00 B
    - 01 H
    - 10 S
    - 11 D
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = SInt(Elem[operand1, e + 0, esize]);
      element2 = SInt(Elem[operand1, e + 1, esize]);
    else
      element1 = SInt(Elem[operand2, e - 1, esize]);
      element2 = SInt(Elem[operand2, e + 0, esize]);
    integer res = Min(element1, element2);
    Elem[result, e, esize] = res<esize-1:0>;
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09.xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMINV

Signed minimum reduction to scalar.

Signed minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum signed integer for the element size.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>size</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>Pg</td>
</tr>
<tr>
<td>24</td>
<td>Zn</td>
</tr>
<tr>
<td>23</td>
<td>Vd</td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

SMINV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        minimum = Min(minimum, element);
V[d] = minimum<esize-1:0>;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  ◦ The values of the NZCV flags.
SMLALB (vectors)

Signed multiply-add long to accumulator (bottom).

Multiply the corresponding even-numbered signed elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
0 1 0 0 0 1 0 0 | size | 0 0 1 0 1 0 0 0 | Zm | Zn | Zda
S U T
```

SMLALB <Zda>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << Uint(size);
integer n = Uint(Zn);
integer m = Uint(Zm);
integer da = Uint(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
text integer elements = VL DIV esize;
text bits(VL) operand1 = Z[n];
text bits(VL) operand2 = Z[m];
text bits(VL) result = Z[da];
for e = 0 to elements-1
    text integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    text integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    text bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;
Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMLALB (indexed)

Signed multiply-add long to accumulator (bottom, indexed).

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

**32-bit**

```
0 1 0 0 0 1 0 0 | 1 0 1 i3h | Zm 1 0 0 0 | i3l 0 | Zn 0 Zda
```


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

**64-bit**

```
0 1 0 0 0 1 0 0 | 1 1 i2h | Zm 1 0 0 0 | i2l 0 | Zn 0 Zda
```

SMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

**Assembler Symbols**

- `<Zda>`: Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>`: Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>`: For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
- `<imm>`: For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMLALT (vectors)

Signed multiply-add long to accumulator (top).

Multiply the corresponding odd-numbered signed elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
+--------------------------------+-------------------+---------+---------+-------+
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 1                   |        |        |
| 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 1                   | Zm     | Zn     | Zda    |
|                                S U T                                      |
```

SMLALT \(<Zda>\).<T>, \(<Zn>\).<Tb>, \(<Zm>\).<Tb>

if !\(\text{HaveSVE2}()\) then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 \(<\text{UInt}(\text{size})\);
integer n = \(\text{UInt}(\text{Zn})\);
integer m = \(\text{UInt}(\text{Zm})\);
integer da = \(\text{UInt}(\text{Zda})\);

Assembler Symbols

\(<Zda>\) Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.
\(<Tb>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
\text{CheckSVEEnabled}();
integer elements = VL \(\text{DIV} \) esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer element1 = \(\text{SInt}(\text{Elem}(\text{operand1}, 2*e + 1, \text{esize DIV 2}))\);
    integer element2 = \(\text{SInt}(\text{Elem}(\text{operand2}, 2*e + 1, \text{esize DIV 2}))\);
    bits(esize) product = (element1 * element2)<\text{esize-1:0}>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;
Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMLALT (indexed)

Signed multiply-add long to accumulator (top, indexed).

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1 0 1 i3h Zm 1 0 0 0</td>
<td>i3l 1 Zn Zda</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>1 1 1 i2h Zm 1 0 0 0</td>
<td>i2l 1 Zn Zda</td>
</tr>
</tbody>
</table>

SMLALT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

### Assembler Symbols

- `<Zda>`
  Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

- `<Zn>`
  Is the name of the first source scalable vector register, encoded in the “Zn” field.

- `<Zm>`
  For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

- `<imm>`
  For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMLSLB (vectors)

Signed multiply-subtract long from accumulator (bottom).

Multiply the corresponding even-numbered signed elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  |

SMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer n = \texttt{UInt}(Zn);
integer m = \texttt{UInt}(Zm);
integer da = \texttt{UInt}(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

\texttt{CheckSVEEnabled}();
integer elements = \texttt{VL} \texttt{DIV} esize;
bits(\texttt{VL}) operand1 = \texttt{Z}[n];
bits(\texttt{VL}) operand2 = \texttt{Z}[m];
bits(\texttt{VL}) result = \texttt{Z}[da];
for e = 0 to elements-1
    integer element1 = \texttt{SInt(Elem}(operand1, 2*e + 0, esize \texttt{DIV} 2));
    integer element2 = \texttt{SInt(Elem}(operand2, 2*e + 0, esize \texttt{DIV} 2));
    bits(esize) product = (element1 * element2)<esize-1:0>;
    \texttt{Elem}[result, e, esize] = \texttt{Elem}[result, e, esize] - product;
\texttt{Z}[da] = result;

Operational information

If PSTATE.DIT is 1:
  \begin{itemize}
    \item The execution time of this instruction is independent of:
      \begin{itemize}
        \item The values of the data supplied in any of its registers.
        \item The values of the NZCV flags.
      \end{itemize}
  \end{itemize}
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMLSLB (indexed)

Signed multiply-subtract long from accumulator (bottom, indexed).

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 1 i3h Zm 1 0 1 0 i3l 0 Zn Zda</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt; S U T</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 1 i2h Zm 1 0 1 0 i2l 0 Zn Zda</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt; S U T</td>
</tr>
</tbody>
</table>

SMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

**Assembler Symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMLSLT (vectors)

Signed multiply-subtract long from accumulator (top).

Multiply the corresponding odd-numbered signed elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------------------------|------------------|------------------|------------------|
| 0 1 0 0 1 0 0 | 0 | Zm | 0 1 0 | 1 0 | 1 | Zn | Zda |

SMLSLT <Zda>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

- <Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- <T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- <Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
- <Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- <Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
  bits(esize) product = (element1 * element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] - product;
Z[da] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

  • The MOVPRFX instruction must be unpredicated.
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMLSLT (indexed)

Signed multiply-subtract long from accumulator (top, indexed).

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | i3h| Zm | 1  | 0  | 1  | 0  | i3l| 1  | Zn |    |    |    |    |    |    |    |    |    |    |    |    |

```
```

if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | i2h| Zm | 1  | 0  | 1  | 0  | i2l| 1  | Zn |    |    |    |    |    |    |    |    |    |    |    |    |    |

```
```

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bite(VL) operand1 = Z[n];
bite(VL) operand2 = Z[m];
bite(VL) result = Z[da];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bite(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMMLA

Signed integer matrix multiply-accumulate.

The signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of signed 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Zm</td>
<td>1 0</td>
<td>0 1</td>
<td>1 0</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(128) op1, op2;
bits(128) res, addend;
for s = 0 to segments-1
    op1 = Elem[operand1, s, 128];
    op2 = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
SMULH (predicated)

Signed multiply returning high half (predicated).

Widening multiply signed integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{cccccccccccccccccccccccccc}
\text{31} & \text{30} & \text{29} & \text{28} & \text{27} & \text{26} & \text{25} & \text{24} & \text{23} & \text{22} & \text{21} & \text{20} & \text{19} & \text{18} & \text{17} & \text{16} & \text{15} & \text{14} & \text{13} & \text{12} & \text{11} & \text{10} & \text{9} & \text{8} & \text{7} & \text{6} & \text{5} & \text{4} & \text{3} & \text{2} & \text{1} & \text{0} \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \text{size} & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \text{Pg} & \text{Zm} & \text{Zdn} \\
\end{array}
\]

SMULH \(<\text{Zdn}>..<\text{T}>, \ <\text{Pg}>/\text{M}, \ <\text{Zdn}>..<\text{T}>, \ <\text{Zm}>.<\text{T}>

if !\text{HaveSVE}() then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer g = \text{UInt}(\text{Pg});
integer dn = \text{UInt}(\text{Zdn});
integer m = \text{UInt}(\text{Zm});
boolean unsigned = FALSE;

Assembler Symbols

\(<\text{Zdn}>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
\(<\text{T}>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = \text{VL} \text{DIV} \text{esize};
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = \text{Int(Elem[operand1, e, esize], unsigned)};
  integer element2 = \text{Int(Elem[operand2, e, esize], unsigned)};
  if Elem[mask, e, esize] == '1' then
    integer product = (element1 * element2) >> \text{esize};
    Elem[result, e, esize] = product<\text{esize}-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
\Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SMULH (unpredicated)

Signed multiply returning high half (unpredicated).

Widening multiply signed integer values of all elements of the first source vector by corresponding elements of the second source vector and place the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   | 1   | 0   | 0   | 1   | Zm | 0   | 1   | 1   | 0   | Zn | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | Zd | 0   | U |
```

SMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVEz() then UNDEFINED;
iinteger esize = 8 << UInt(size);
iinteger n = UInt(Zn);
iinteger m = UInt(Zm);
iinteger d = UInt(Zd);
boolean unsigned = FALSE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
iinteger elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer product = (element1 * element2) >> esize;
    Elem[result, e, esize] = product<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SMULLB (vectors)

Signed multiply long (bottom).

Multiply the corresponding even-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```assembly
if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler Symbols**

- `<Zd>`: Is the name of the destination scalable vector register, encoded in the “Zd” field.
- `<T>`: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>`: Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>`: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>`: Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```assembly
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem(operand1, 2*e + 0, esize DIV 2));
    integer element2 = SInt(Elem(operand2, 2*e + 0, esize DIV 2));
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMULLB (indexed)

Signed multiply long (bottom, indexed).

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 0 0 1 0 0 1 0 1 1 | Zm 1 1 0 0 | Zn | Zd |


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;

### 64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 0 0 1 0 0 1 0 1 1 | Zm 1 1 0 0 | Zn | Zd |

**SMULLB**<Zd>.D, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;

### Assembler Symbols

**<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.

**<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.

**<Zm>** For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

**<imm>** For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMULLT (vectors)

Signed multiply long (top).

Multiply the corresponding odd-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 0   | 1   | 0   | 1   | Zm  | 0   | 1   | 1   | 1   | 0   | 1   | Zn  | Zd  |

**SMULLT <Zd>..<T>, <Zn>..<Tb>, <Zm>..<Tb>**

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the “Zd” field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the first source scalable vector register, encoded in the “Zn” field.
- **<Tb>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- **<Zm>** Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

- **CheckSVEEnabled();**
- integer elements = VL DIV esize;
- bits(VL) operand1 = Z[n];
- bits(VL) operand2 = Z[m];
- bits(VL) result;
- for e = 0 to elements-1
  - integer element1 = SInt(Elem(operand1, 2*e + 1, esize DIV 2));
  - integer element2 = SInt(Elem(operand2, 2*e + 1, esize DIV 2));
  - integer res = element1 * element2;
  - Elem[result, e, esize] = res<esize-1:0>;
- Z[d] = result;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SMULLT (indexed)

Signed multiply long (top, indexed).

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified
element in the corresponding second source vector segment, and place the results in the overlapping double-width
elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element
position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per
128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i3h</td>
<td>Zm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>i3l</td>
<td>1</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size<1>size<0>  U  T


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;

64-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>i2h</td>
<td>Zm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>i2l</td>
<td>1</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

size<1>size<0>  U  T

SMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the
"Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the
"Zm" field.
<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

CheckSVEEnabled();
integer elements = $VL$ DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits($VL$) operand1 = $Z[n]$;
bits($VL$) operand2 = $Z[m]$;
bits($VL$) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

$Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SPLICE

Splice two vectors under predicate control.

Copy the first active to last active elements (inclusive) from the first source vector to the lowest-numbered elements of the result. Then set any remaining elements of the result to a copy of the lowest-numbered elements from the second source vector. The result is placed destructively in the destination and first source vector, or constructively in the destination vector.

It has encodings from 2 classes: Constructive and Destructive

Constructive

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 size 1 0 1 1 0 1 1 0 0 Pg Zn Zd
```

```
SPLICE <Zd>.<T>, <Pg>, { <Zn1>.<T>, <Zn2>.<T> }
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dst = UInt(Zd);
integer s1 = UInt(Zn);
integer s2 = (s1 + 1) MOD 32;
```

Destructive

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 size 1 0 1 1 0 0 1 0 0 Pg Zm Zdn
```

```
SPLICE <Zdn>.<T>, <Pg>, { <Zdn>.<T>, <Zm>.<T> }
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dst = UInt(Zdn);
integer s1 = dst;
integer s2 = UInt(Zm);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

```
size <T>
00 0
01 1
10 S
11 D
```

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[s1];
bits(VL) operand2 = Z[s2];
bits(VL) result;
integer x = 0;
boolean active = FALSE;
integer lastnum = LastActiveElement(mask, esize);

if lastnum >= 0 then
    for e = 0 to lastnum
        active = active || ElemP[mask, e, esize] == '1';
        if active then
            Elem[result, x, esize] = Elem[operand1, e, esize];
            x = x + 1;

elements = elements - x - 1;
for e = 0 to elements
    Elem[result, x, esize] = Elem[operand2, e, esize];
    x = x + 1;

Z[dst] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQABS

Signed saturating absolute value.

Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). Inactive elements in the destination vector register remain unmodified.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 0 0 0 1 0 1  |  Pg  |  Zn  |  Zd  |
Q
```

**SQABS** <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in "size":

  ```
  size  <T>
  00   B
  01   H
  10   S
  11   D
  ```

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

- `CheckSVEEnabled();`
- integer elements = VL DIV esize;
- bits(PL) mask = P[g];
- bits(VL) operand = Z[n];
- bits(VL) result = Z[d];

  for e = 0 to elements-1
    integer element = SInt(Elem[operand, e, esize]);
    if ElemP[mask, e, esize] == '1' then
      element = Abs(element);
    Elem[result, e, esize] = SignedSat(element, esize);

  Z[d] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQADD (vectors, predicated)

Signed saturating addition (predicated).

Add active signed elements of the first source vector to corresponding signed elements of the second source vector
and destructively place the results in the corresponding elements of the first source vector. Each result element is
saturated to the N-bit element’s signed integer range \(-2^{(N-1)} to 2^{(N-1)}\)-1. Inactive elements in the destination vector
register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 | 0 1 1 0 0 0 1 0 0 | Pg | Zm | Zdn

SQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = SInt(Sat(element1 + element2, esize, unsigned));
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register
  and source element size as this instruction.
• The **MOVPRFX** instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQADD (immediate)

Signed saturating add immediate (unpredicated).

Signed saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)}) - 1$. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280. The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is “#<imm8>, LSL #8”. However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as “#0, LSL #8”.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 0 1 0 0 1 0 1 0 0 1 0 0 1 1 sh imm8 Zdn
```

**SQADD <Zdn>..<T>, <Zdn>..<T>, #<imm>{, <shift>}

if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], ·) = SatQ(element1 + imm, esize, unsigned);
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQADD (vectors, unpredicated)

Signed saturating add vectors (unpredicated).

Signed saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \text{size} & 1 & Zm & 0 & 0 & 0 & 1 & 0 & 0 & Zn & 0 & 0 & 0 & 1 & Zd & U
\end{array}
\]

SQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 \ll \text{UInt}(\text{size});
integer n = \text{UInt}(Zn);
integer m = \text{UInt}(Zm);
integer d = \text{UInt}(Zd);
boolean unsigned = FALSE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL \div \text{esize};
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = \text{Int}(\text{Elem}[\text{operand1}, e, \text{esize}], \text{unsigned});
  integer element2 = \text{Int}(\text{Elem}[\text{operand2}, e, \text{esize}], \text{unsigned});
  \text{Elem}[\text{result}, e, \text{esize}, -] = \text{SatQ}(\text{element1} + \text{element2}, \text{esize}, \text{unsigned});
Z[d] = result;
Saturating complex integer add with rotate.

Add the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±j beforehand. Destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & \text{rot} & \text{Zm} & \text{Zdn} \\
\end{array}
\]

SQCADD <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>

```hs
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in “rot”:

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#90</td>
</tr>
<tr>
<td>1</td>
<td>#270</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for p = 0 to pairs-1
    integer acc_r  = SInt(Elem[operand1, 2 * p + 0, esize]);
    integer acc_i  = SInt(Elem[operand1, 2 * p + 1, esize]);
    integer elt2_r = SInt(Elem[operand2, 2 * p + 0, esize]);
    integer elt2_i = SInt(Elem[operand2, 2 * p + 1, esize]);
    if sub_i then
        acc_r = acc_r - elt2_i;
        acc_i = acc_i + elt2_r;
    if sub_r then
        acc_r = acc_r + elt2_i;
        acc_i = acc_i - elt2_r;

    Elem[result, 2 * p + 0, esize] = SignedSat(acc_r, esize);
    Elem[result, 2 * p + 1, esize] = SignedSat(acc_i, esize);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count.

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>pattern</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
<td>sf</td>
<td>D</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQDECB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

### 64-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>pattern</td>
<td>Rdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
<td>sf</td>
<td>D</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQDECB <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

### Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

SQDECB
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>10000</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>10001</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>10010</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>10101</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>10110</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>11001</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>11010</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>11100</td>
<td>#ui16x1</td>
</tr>
<tr>
<td>11110</td>
<td>MUL4</td>
</tr>
<tr>
<td>11111</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDECD (scalar)**

Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: [32-bit](#) and [64-bit](#).

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | imm4 | 1  | 1  | 1  | 1  | 0  | pattern | Rdn |

SQDECD `<Xdn>`, `<Wdn>{{, <pattern>{{, MUL #<imm>{{}}}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

### 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | imm4 | 1  | 1  | 1  | 1  | 1  | 0  | pattern | Rdn |

SQDECD `<Xdn>{{, <pattern>{{, MUL #<imm>{{}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

**Assembler Symbols**

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECD (vector)

Signed saturating decrement vector by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by
an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The
results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than
Undefined Instruction exception.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0 1 0 0 0 0 0 0 0 1 0 0
size<1>size<0> imm4 pattern Zdn
```

SQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECH (scalar)

Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
</tr>
</tbody>
</table>

SQDECH <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}</p>

```java
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
</tr>
</tbody>
</table>

SQDECH <Xdn>{, <pattern>{, MUL #<imm>}}</p>

```java
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

### Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

**SQDECH (scalar)**
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11001</td>
<td>MUL4</td>
</tr>
<tr>
<td>11100</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<immediate multiplier> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECH (vector)

Signed saturating decrement vector by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>pattern</td>
<td>Zdn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01010</td>
<td>VL16</td>
</tr>
<tr>
<td>01101</td>
<td>VL16</td>
</tr>
<tr>
<td>01110</td>
<td>VL32</td>
</tr>
<tr>
<td>01111</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx01</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx10</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

\[ \text{CheckSVEEnabled();} \]
\[ \text{integer elements = \text{VL} \text{ DIV esize};} \]
\[ \text{integer count = \text{DecodePredCount}(pat, esize);} \]
\[ \text{bits(\text{VL}) operand1 = \text{Z}[dn];} \]
\[ \text{bits(\text{VL}) result;} \]

\[
\text{for } e = 0 \text{ to elements-1} \\
\quad \text{integer element1 = \text{Int(Elem)[operand1, e, esize], unsigned);} \\
\quad (\text{Elem}[\text{result}, e, \text{esize}], -) = \text{SatQ(element1 - (count * imm), esize, unsigned);} \\
\]
\[ \text{Z[dn] = result;} \]

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDECP (scalar)

Signed saturating decrement scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 size 1 0 1 0 1 0 0 1 0 0 Pm Rdn
```

SQDECP <Xdn>, <Pm>.<T>, <Wdn>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 size 1 0 1 0 1 0 0 1 0 1 0 0 1 1 0 Pm Rdn
```

SQDECP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Pm> Is the name of the source scalable predicate register, encoded in the “Pm” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
CheckSVEEnabled();
ineger elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(ssize) operand1 = \texttt{X}[dn];
bits(PL) operand2 = \texttt{P}[m];
bits(ssize) result;
ineger count = 0;

for e = 0 to elements-1
    if \texttt{ElemP}[\texttt{operand2}, e, \texttt{esize}] == '1' then
        count = count + 1;

integer element = \texttt{Int}(\texttt{operand1}, \texttt{unsigned});
(result, -) = \texttt{SatQ}(element - count, ssize, unsigned);
\texttt{X}[dn] = \texttt{Extend}(result, 64, unsigned);
SQDECP (vector)

Signed saturating decrement vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Pm  | Zdn |

SQDECP <Zdn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the “Pm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDECW (scalar)

Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 imm4 1 1 1 1 1 1 1 0 pattern Rdn

size<1>size<0> sf D U
```

SQDECW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 imm4 1 1 1 1 1 1 1 1 0 pattern Rdn

size<1>size<0> sf D U
```

SQDECW <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11011</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDECW (vector)

Signed saturating decrement vector by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Unsaturated 32-bit signed decrement:

\[
\text{dst} = \text{src} - \text{imm} \times (\text{dn} - 1) \times \text{pattern}
\]

\[
= \begin{cases} 
0 & \text{if } \text{pattern} = \text{ALL} \\
\text{imm} \times (\text{dn} - 1) & \text{else}
\end{cases}
\]

if \(\text{HaveSVE}()\) then UNDEFINED;

\[
\text{integer esize} = 32;
\text{integer dn} = \text{UInt}(\text{Zdn});
\text{bits(5)} \text{ pat} = \text{pattern};
\text{integer imm} = \text{UInt}(\text{imm4}) + 1;
\text{boolean unsigned} = \text{FALSE};
\]

Assembler Symbols

<Zdn>  Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<pattern>  Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm>  Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bites(VL) operand1 = Z[dn];
bites(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQDMLALB (vectors)**

Signed saturating doubling multiply-add long to accumulator (bottom).

Multiply then double the corresponding even-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)}-1)$. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)}-1)$. This instruction is unpredicated.

```
0 1 0 0 0 1 0 0  size  0  Zm  0 1 1 0 0 0 0  Zn  Zda
S  T
```

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

1. `CheckSVEEnabled();`
2. `integer elements = VL DIV esize;`
3. `bits(VL) operand1 = Z[n];`
4. `bits(VL) operand2 = Z[m];`
5. `bits(VL) result = Z[da];`
6. `for e = 0 to elements-1`
   - `integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);`
   - `integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);`
   - `integer element3 = SInt(Elem[result, e, esize]);`
   - `integer product = SInt(SignedSat(2 * element1 * element2, esize));`
   - `Elem[result, e, esize] = SignedSat(element3 + product, esize);`
7. `Z[da] = result;`
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQDMLALB (indexed)**

Signed saturating doubling multiply-add long to accumulator (bottom, indexed).

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | i3h | Zm | 0  | 0  | 1  | 0  | i3l | Zn | Zda |
```

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
</table>


```
if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### 64-bit

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | i2h | Zm | 0  | 0  | 1  | 0  | i2l | Zn | Zda |
```

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
</table>

SQDMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

### Assembler Symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
- `<imm>` For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \div (2 \times \text{esize});
integer eltspersegment = 128 \div (2 \times \text{esize});
bits(\texttt{VL}) operand1 = \texttt{Z}[n];
bits(\texttt{VL}) operand2 = \texttt{Z}[m];
bits(\texttt{VL}) result = \texttt{Z}[da];

for \( e = 0 \) to elements - 1
  integer \( s = e - (e \mod \text{eltspersegment}); \)
  integer element1 = \texttt{SInt}(\texttt{Elem}[\text{operand1}, 2 \times e + \text{sel}, \text{esize}]);
  integer element2 = \texttt{SInt}(\texttt{Elem}[\text{operand2}, 2 \times s + \text{index}, \text{esize}]);
  integer element3 = \texttt{SInt}(\texttt{Elem}[\text{result}, e, 2 \times \text{esize}]);
  integer product = \texttt{SInt(\texttt{SignedSat}(2 \times \text{element1} \times \text{element2}, 2 \times \text{esize}))};
  integer res = element3 + product;
  \texttt{Elem}[result, e, 2 \times \text{esize}] = \texttt{SignedSat}(res, 2 \times \text{esize});

\texttt{Z}[da] = result;

Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is UNPREDICTABLE:

- The \texttt{MOVPRFX} instruction must be unpredicated.
- The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQDMLALBT**

Signed saturating doubling multiply-add long to accumulator (bottom × top).

Multiply then double the corresponding even-numbered signed elements of the first and odd-numbered signed elements of the second source vector. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

![Binary representation of SQDMLALBT](image)

SQDMLALBT `<Zda>..<T>, <Zn>..<Tb>, <Zm>..<Tb>`

```asm
if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 1;
```

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:
  ```
  size   <T>
  00     RESERVED
  01     H
  10     S
  11     D
  ```
- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` Is the size specifier, encoded in “size”:
  ```
  size   <Tb>
  00     RESERVED
  01     B
  10     H
  11     S
  ```
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```asm
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
  integer element3 = SInt(Elem[result, e, esize]);
  integer product = SInt(SignedSat(2 * element1 * element2, esize));
  Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da] = result;
```

SQDMLALBT
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDMLALT (vectors)

Signed saturating doubling multiply-add long to accumulator (top).

Multiply then double the corresponding odd-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). This instruction is unpredicated.

### Assembler Symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

### Operation

```assembly
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 + product, esize);
Z[da] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed saturating doubling multiply-add long to accumulator (top, indexed).

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: **32-bit** and **64-bit**

**32-bit**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 1 0 1 i3h Zm 0 0 1 0 i3l 1 Zn Zda
```


```java
if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

**64-bit**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 1 1 i2h Zm 0 0 1 0 i2l 1 Zn Zda
```


```java
if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
  
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- `<imm>` For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 + product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDMLSLB (vectors)

Signed saturating doubling multiply-subtract long from accumulator (bottom).

Multiply then double the corresponding even-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

```
0 1 0 0 1 0 0 | size | 0 | Zm | 0 1 1 0 1 0 | Zn | Zda
```

```
SQDMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 0;

**Assembler Symbols**

- **<Zda>** Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the first source scalable vector register, encoded in the “Zn” field.
- **<Tb>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- **<Zm>** Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

- **CheckSVEEnabled();**
- integer elements = VL DIV esize;
- bits(VL) operand1 = Z[n];
- bits(VL) operand2 = Z[m];
- bits(VL) result = Z[da];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQDMLSLB (indexed)**

Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed).

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i3h</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i3l</td>
<td>0</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
<td>S</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

### 64-bit

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i2h</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>i2l</td>
<td>0</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
<td>S</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQDMLSLB <Zda>.D, <Zn>.S, <Zm>.S[iimm]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
  
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- `<imm>` For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements - 1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 - product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDMLSLBT

Signed saturating doubling multiply-subtract long from accumulator (bottom × top).

Multiply then double the corresponding even-numbered signed elements of the first and odd-numbered signed elements of the second source vector. Each intermediate value is saturated to the double-width N-bit value’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

### Assembler Symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th><code>size</code></th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

### Operation

- `CheckSVEEnabled();`
- `integer elements = VL DIV esize;`
- `bits(VL) operand1 = Z[n];`
- `bits(VL) operand2 = Z[m];`
- `bits(VL) result = Z[da];`

for `e = 0` to `elements-1`

- `integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);`
- `integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);`
- `integer element3 = SInt(Elem[result, e, esize]);`
- `integer product = SInt(SignedSat(2 * element1 * element2, esize));`
- `Elem[result, e, esize] = SignedSat(element3 - product, esize);`

`Z[da] = result;`
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQDMLSLT (vectors)**

Signed saturating doubling multiply-subtract long from accumulator (top).

Multiply then double the corresponding odd-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 | size | 0 | Zm | 0 1 1 0 | 1 1 | Zn | Zda
```

**SQDMLSLT** `<Zda>,<T>, <Zn>,<Tb>, <Zm>,<Tb>`

if `!HaveSVE2()` then UNDEFINED;
if `size == '00'` then UNDEFINED;
integer `esize = 8 << UInt(size);`
integer `n = UInt(Zn);`
integer `m = UInt(Zm);`
integer `da = UInt(Zda);`
integer `sell = 1;`
integer `sel2 = 1;`

**Asmmler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```
CheckSVEEnabled();
integer `elements = VL DIV esize;`
bits(VL) `operand1 = Z[n];`
bits(VL) `operand2 = Z[m];`
bits(VL) `result = Z[da];`

for e = 0 to elements-1
  integer `element1 = SInt(Elem[operand1, 2 * e + sell, esize DIV 2]);`
  integer `element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);`
  integer `element3 = Int(Elem[result, e, esize]);`
  integer `product = SInt(SignedSat(2 * element1 * element2, esize));`
  Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da] = result;
```
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed saturating doubling multiply-subtract long from accumulator (top, indexed).

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 0 1 0 0 1 0 1 i3h Zm 0 0 1 1 i3l 1 Zn Zda</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 0 1 0 0 1 1 i2h Zm 0 0 1 1 i2l 1 Zn Zda</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

### Assembler Symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
  
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- `<imm>` For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements - 1
  integer s = e - (e MOD eltspersegment);
  integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
  integer element3 = SInt(Elem[result, e, 2*esize]);
  integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
  integer res = element3 - product;
  Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQDMULH (vectors)

Signed saturating doubling multiply high (unpredicated).

Multiply then double the corresponding signed elements of the first and second source vectors, and place the most significant half of the results in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)}-1$. This instruction is unpredicated.

```
0 0 0 0 0 1 0 0 | size | 1     | Zm   | 0 1 1 1 0 0 | Zn   | Zd   
```

```
SQDMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

```c
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

size <T>

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res >> esize, esize);
Z[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMULH (indexed)**

Signed saturating doubling multiply high (indexed).

Multiply all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, double and place the most significant half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element’s signed integer range $-2^{(N-1)}$ to $2^{(N-1)}-1$.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

It has encodings from 3 classes: **16-bit**, **32-bit** and **64-bit**

### 16-bit

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Zm</td>
<td></td>
<td>Zd</td>
<td></td>
</tr>
</tbody>
</table>

If `HaveSVE2()` then UNDEFINED.

```assembly
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

### 32-bit

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>i2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Zm</td>
<td></td>
<td>Zd</td>
</tr>
</tbody>
</table>

Size<1>Size<0>

If `HaveSVE2()` then UNDEFINED.

```assembly
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

### 64-bit

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Zm</td>
<td></td>
</tr>
</tbody>
</table>

Size<1>Size<0>

If `HaveSVE2()` then UNDEFINED.

```assembly
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

### Assembler Symbols

 `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res >> esize, esize);
Z[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMULLB (vectors)

Signed saturating doubling multiply long (bottom).

Multiply the corresponding even-numbered signed elements of the first and second source vectors, double and place the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

\[
\begin{array}{ccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\begin{array}{ccccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{array}

\[<Zd>, <T>, <Zn>, <Tb>, <Zm>, <Tb>\]

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer n = \text{UInt}(Zn);
integer m = \text{UInt}(Zm);
integer d = \text{UInt}(Zd);

Assembler Symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<Tb>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\(\text{CheckSVEEnabled}();\)
integer elements = \text{VL} \text{DIV} \text{esize};
bits(\text{VL}) \text{operand1} = \text{Z}[n];
bits(\text{VL}) \text{operand2} = \text{Z}[m];
bits(\text{VL}) \text{result};
for e = 0 to elements-1
\text{integer element1} = \text{SInt}(\text{Elem}[\text{operand1}, 2*e + 0, \text{esize} \text{DIV} 2]);
\text{integer element2} = \text{SInt}(\text{Elem}[\text{operand2}, 2*e + 0, \text{esize} \text{DIV} 2]);
\text{integer res} = 2 * \text{element1} * \text{element2};
\text{Elem}[\text{result}, e, \text{esize}] = \text{SignedSat}(\text{res}, \text{esize});
\text{Z}[d] = \text{result};
SQDMULLB (indexed)

Signed saturating doubling multiply long (bottom, indexed).

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and the specified element in the corresponding second source vector segment, and place the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 1 0 1 i3h</td>
</tr>
</tbody>
</table>

size<1>|size<0> | T


if !HaveSVE2() then UNDEFINED;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 1 1 0 1 i2h</td>
</tr>
</tbody>
</table>

size<1>|size<0> | T

SQDMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;

Assembler Symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.

\(<Zm>\) For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

\(<imm>\) For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQDMULLT (vectors)

Signed saturating doubling multiply long (top).

Multiply the corresponding odd-numbered signed elements of the first and second source vectors, double and place the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

```
0 1 0 0 0 1 0 1 | size | Zm 0 1 1 0 0 1 | Zn | Zd
```

SQDMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

If !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt (size);
integer n = UInt (Zn);
integer m = UInt (Zm);
integer d = UInt (Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res, esize);

Z[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQDMULLT (indexed)**

Signed saturating doubling multiply long (top, indexed).

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified element in the corresponding second source vector segment, and place the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element’s signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | i3h | Zm | 1 | 1 | 1 | 0 | i3l | 1 | Zn | Zd |

\[
\text{SQDMULLT } \langle Zd \rangle.S, \langle Zn \rangle.H, \langle Zm \rangle.H[\langle \text{imm} \rangle]
\]

if !\(\text{HaveSVE2}()\) then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;

### 64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | i2h | Zm | 1 | 1 | 1 | 0 | i2l | 1 | Zn | Zd |

\[
\text{SQDMULLT } \langle Zd \rangle.D, \langle Zn \rangle.S, \langle Zm \rangle.S[\langle \text{imm} \rangle]
\]

if !\(\text{HaveSVE2}()\) then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;

### Assembler Symbols

- **\(<Zd>****:** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **\(<Zn>****:** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **\(<Zm>****:** For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- **\(<\text{imm}>****:** For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltsperegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltsperegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[d] = result;
```

SQINCB

Signed saturating increment scalar by multiple of 8-bit predicate constraint element count.

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

**32-bit**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 0 1 0 1 1 1 1 0 0 pattern Rdn
size<1>size<0> sf D U
```

SQINCB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

**64-bit**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 0 1 0 1 1 1 1 0 0 pattern Rdn
size<1>size<0> sf D U
```

SQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

**Assembler Symbols**

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

SQINCB
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1101</td>
<td>MUL4</td>
</tr>
<tr>
<td>1110</td>
<td>MUL3</td>
</tr>
<tr>
<td>1111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

### Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQINCD (scalar)

Signed saturating increment scalar by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

SQINCD <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

SQINCD <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11110</td>
<td>MUL4</td>
</tr>
<tr>
<td>11111</td>
<td>MUL3</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
SQINCD (vector)

Signed saturating increment vector by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 1 0 0 1 1 1 0 imm4 1 1 0 0 0 0 pattern Zdn |

SQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}

```asm
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

- **<Zdn>** Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
- **<pattern>** Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

- **<imm>** Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} esize;
integer count = \texttt{DecodePredCount}(\texttt{pat}, \texttt{esize});
bits(\texttt{VL}) operand1 = \texttt{Z}[dn];
bits(\texttt{VL}) result;

for e = 0 to elements-1
  integer element1 = \texttt{Int(Elem)}[operand1, e, esize], unsigned);
  (\texttt{Elem}[result, e, esize], -) = \texttt{SatQ}(element1 + (count * imm), esize, unsigned);
\texttt{Z}[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is \texttt{UNPREDICTABLE}:

- The \texttt{MOVPRFX} instruction must be unpredicated.
- The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQINCH (scalar)**

Signed saturating increment scalar by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 0</td>
<td>1 1 0 imm4</td>
<td>1 1 1 1 0 0 pattern Rdn</td>
</tr>
</tbody>
</table>

* size<1>size<0> sf D U

SQINCH <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UINT(Rdn);
bits(5) pat = pattern;
integer imm = UINT(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>23 22 21 20 19 18 17 16</th>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 0</td>
<td>1 1 1 imm4</td>
<td>1 1 1 1 0 0 pattern Rdn</td>
</tr>
</tbody>
</table>

* size<1>size<0> sf D U

SQINCH <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UINT(Rdn);
bits(5) pat = pattern;
integer imm = UINT(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

### Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

SQINCH (scalar)
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>PW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>1x010</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10x01</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Signed saturating increment vector by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

### Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQINCP (scalar)**

Signed saturating increment scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 0 0 1 0 1 | size | 1 0 1 0 | 0 0 1 0 0 1 0 | Pm | Rdn |

SQINCP <Xdn>, <Pm>.<T>, <Wdn>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;

### 64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 0 0 1 0 1 | size | 1 0 1 0 | 0 0 1 0 0 1 1 | 0 | Pm | Rdn |

SQINCP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;

**Assembler Symbols**

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

<Pm> Is the name of the source scalable predicate register, encoded in the “Pm” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
SQINCP (vector)

Signed saturating increment vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | size | 1 0 1 0 | 0 0 1 0 0 0 0 0 | 0 0 | Pm | Zdn
D U
```

SQINCP <Zdn>..<T>, <Pm>..<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;
for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQINCW (scalar)

Signed saturating increment scalar by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register’s signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 1 0 1 1 1 1 0 0 | pattern | Rdn |
| size<1>size<0> | sf | D | U |

SQINCW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;

64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 0 1 1 1 1 1 1 0 0 | pattern | Rdn |
| size<1>size<0> | sf | D | U |

SQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQINCW (vector)

Signed saturating increment vector by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---------------|----------------|----------------|----------------|
| 0 0 0 0 0 0 1 0 0 1 0 | imm4 | 1 1 0 0 | Zdn |

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

size<1>size<0> D U

SQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01010</td>
<td>VL16</td>
</tr>
<tr>
<td>01011</td>
<td>VL32</td>
</tr>
<tr>
<td>01100</td>
<td>VL64</td>
</tr>
<tr>
<td>01101</td>
<td>VL128</td>
</tr>
<tr>
<td>01110</td>
<td>VL256</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQNEG**

Signed saturating negate.

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQNEG `<Zd>`.<`T`>, `<Pg>`/<M>, `<Zn>`.<`T`>

if !`HaveSVE2()` then UNDEFINED;

integer esize = 8 << `UInt`(size);
integer g = `UInt`(Pg);
integer n = `UInt`(Zn);
integer d = `UInt`(Zd);

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

`CheckSVEEnabled();`

integer elements = `VL` DIV esize;

`bits(VL)` mask = P[g];
`bits(VL)` operand = Z[n];

`bits(VL)` result = Z[d];

for e = 0 to elements-1

    integer element = `SInt`(Elem[operand, e, esize]);
    if `ElemP`[mask, e, esize] == '1' then
        element = -element;
    `Elem`[result, e, esize] = `SignedSat`(element, esize);

Z[d] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQRDCMLAH (vectors)

Saturating rounding doubling complex integer multiply-add high with rotate.

Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>rot</th>
<th>Zn</th>
<th>Zda</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
<td>0</td>
<td>0 0 1 1</td>
<td>0</td>
<td>0 1</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQRDCMLAH <Zda>,<T>, <Zn>.<T>, <Zm>.<T>, <const>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

<const> Is the const specifier, encoded in “rot”:

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>
Operation

\textbf{CheckSVEEnabled}();

\begin{verbatim}
integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

integer round_const = 1 << (esize-1);
integer res_r, res_i;
for p = 0 to pairs-1
       integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
       integer elt2_a = SInt(Elem[operand2, 2 * p + sel_a, esize]);
       integer elt2_b = SInt(Elem[operand2, 2 * p + sel_b, esize]);
       bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
       bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
       integer product_r = elt1_a * elt2_a;
       integer product_i = elt1_a * elt2_b;
       if sub_r then
          res_r = (SInt(elt3_r) << esize) - 2 * product_r + round_const;
       else
          res_r = (SInt(elt3_r) << esize) + 2 * product_r + round_const;
       if sub_i then
          res_i = (SInt(elt3_i) << esize) - 2 * product_i + round_const;
       else
          res_i = (SInt(elt3_i) << esize) + 2 * product_i + round_const;
       Elem[result, 2 * p + 0, esize] = SignedSat(res_r >> esize, esize);
       Elem[result, 2 * p + 1, esize] = SignedSat(res_i >> esize, esize);
\end{verbatim}

\[ Z[da] = \text{result}; \]

Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is UNPREDICTABLE:

- The \texttt{MOVPRFX} instruction must be unpredicated.
- The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDCMLAH (indexed)

Saturating rounding doubling complex integer multiply-add high with rotate (indexed).

Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation. Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: 16-bit and 32-bit

### 16-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | i2 | Zm | 0  | 1  | 1  | 1  | rot | Zn | Zda |


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | i1 | Zm | 0  | 1  | 1  | 1  | rot | Zn | Zda |

SQRDCMLAH `<Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>`

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in "rot":

<table>
<thead>
<tr>
<th>rot</th>
<th>&lt;const&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#0</td>
</tr>
<tr>
<td>01</td>
<td>#90</td>
</tr>
<tr>
<td>10</td>
<td>#180</td>
</tr>
<tr>
<td>11</td>
<td>#270</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

integer round_const = 1 << (esize-1);
integer res_r, res_i;
for p = 0 to pairs-1
    integer segmentbase = p - (p MOD pairspersegment);
    integer s = segmentbase + index;
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r = elt1_a * elt2_a;
    integer product_i = elt1_a * elt2_b;
    if sub_r then
        res_r = (SInt(elt3_r) << esize) - 2 * product_r + round_const;
    else
        res_r = (SInt(elt3_r) << esize) + 2 * product_r + round_const;
    if sub_i then
        res_i = (SInt(elt3_i) << esize) - 2 * product_i + round_const;
    else
        res_i = (SInt(elt3_i) << esize) + 2 * product_i + round_const;
    Elem[result, 2 * p + 0, esize] = SignedSat(res_r >> esize, esize);
    Elem[result, 2 * p + 1, esize] = SignedSat(res_i >> esize, esize);
Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLAH (vectors)

Signed saturating rounding doubling multiply-add high to accumulator (unpredicated).

Multiply then double the corresponding signed elements of the first and second source vectors, and destructively add the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element’s signed integer range \((-2^{(N-1)})\) to \((2^{(N-1)})-1\). This instruction is unpredicated.

```
0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 0 0 1 1
size Zm Zn Zda
```

SQRDMLAH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) + (2 * element1 * element2);
    Elemp(result, e, esize) = SignedSat((res + round_const) >> esize, esize);
Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQRDMLAH (indexed)**

Signed saturating rounding doubling multiply-add high to accumulator (indexed).

Multiply then double all signed elements within each 128-bit segment of the first source vector and the specified signed element of the corresponding second source vector segment, and destructively add the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\).

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

It has encodings from 3 classes: **16-bit**, **32-bit** and **64-bit**

### 16-bit

[Diagram: 16-bit encoding]

```plaintext
0 1 0 0 0 0 0 0 | i3h | i3l | Zm | 0 0 0 1 0 0 | Zn | Zda
S
```

```plaintext
```

```plaintext
if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 32-bit

[Diagram: 32-bit encoding]

```plaintext
0 1 0 0 0 0 0 0 | 1 | 0 1 | i2 | Zm | 0 0 0 1 0 0 | Zn | Zda
size<1>size<0> S
```

```plaintext
SQRDMLAH <Zda>.S, <Zn>.S, <Zm>.S[<imm>]
```

```plaintext
if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### 64-bit

[Diagram: 64-bit encoding]

```plaintext
0 1 0 0 0 0 1 0 0 | 1 | 1 | i1 | Zm | 0 0 0 1 0 0 | Zn | Zda
size<1>size<0> S
```

```plaintext
```

```plaintext
if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

### Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.
For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, s, esize]);
  integer element3 = SInt(Elem[operand3, e, esize]);
  integer res = (element3 << esize) + (2 * element1 * element2);
  Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);

Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQRDMLSH (vectors)

Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated).

Multiply then double the corresponding signed elements of the first and second source vectors, and destructively subtract the rounded high half of each result from the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)}) - 1\). This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

SQRDMLSH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>
```

If `!HaveSVE2()` then UNDEFINED;

```plaintext
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

**Assembler Symbols**

- `<Zda>`: Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>`: Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>`: Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) - (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);
Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQRDMLSH (indexed)

Signed saturating rounding doubling multiply-subtract high from accumulator (indexed).

Multiply then double all signed elements within each 128-bit segment of the first source vector and the specified signed element of the corresponding second source vector segment, and destructively subtract the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

It has encodings from 3 classes: 16-bit, 32-bit and 64-bit

16-bit

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | i3h| i3l| Zm | 0  | 0  | 0  | 1  | 0  | 1  | Zn |    |    |    |    |    |    |    |    |    |
```


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

32-bit

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | i2 | Zm | 0  | 0  | 0  | 1  | 0  | 1  | Zn |    |    |    |    |    |    |    |    |    |
```

SQRDMLSH <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | Zm | 0  | 0  | 0  | 1  | 0  | 0  | 1  | Zn |    |    |    |    |    |    |    |    |    |
```


if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
integer round_const = 1 << (esize - 1);
for e = 0 to elements - 1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) - (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQRDMULH (vectors)

Signed saturating rounding doubling multiply high (unpredicated).

Multiply then double the corresponding signed elements of the first and second source vectors, and place the most significant rounded half of the result in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). This instruction is unpredicated.

```
0 0 0 0 0 1 0 0 | size 1 | Zm 0 1 1 0 1 | Zn | Zd
R
```

SQRDMULH \(<Zd>\).\(<T>\), \(<Zn>\).\(<T>\), \(<Zm>\).\(<T>\)

if \(!\text{HaveSVE2}()\) then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer n = \text{UInt}(Zn);
integer m = \text{UInt}(Zm);
integer d = \text{UInt}(Zd);

Assembler Symbols

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.
\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.
\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\text{CheckSVEEnabled}();
integer elements = \text{VL} \text{DIV} esize;
bits(\text{VL}) operand1 = Z[n];
bits(\text{VL}) operand2 = Z[m];
bits(\text{VL}) result;
integer round_const = 1 << (esize - 1);
for e = 0 to elements-1
    integer element1 = \text{SInt}(\text{Elem}[operand1, e, esize]);
    integer element2 = \text{SInt}(\text{Elem}[operand2, e, esize]);
    integer res = 2 * element1 * element2;
    \text{Elem}[result, e, esize] = \text{SignedSat}((res + round_const) >> esize, esize);
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMULH (indexed)

Signed saturating rounding doubling multiply high (indexed).

Multiply all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, double and place the most significant rounded half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $2^{(N-1)}-1$.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.

It has encodings from 3 classes: 16-bit, 32-bit and 64-bit

16-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 0 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

SQRDMULH <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm>  For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm>  For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer round_const = 1 << (esize - 1);
for e = 0 to elements - 1
  integer segmentbase = e - (e MOD eltspersegment);
  integer s = segmentbase + index;
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, s, esize]);
  integer res = 2 * element1 * element2;
  Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);
Z[d] = result;
SQRSHL

Signed saturating rounding shift left by vector (predicated).

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

```
|   31  |  30  |  29  |  28  |  27  |  26  |  25  |  24  |  23  |  22  |  21  |  20  |  19  |  18  |  17  |  16  |  15  |  14  |  13  |  12  |  11  |  10  |   9  |   8  |   7  |   6  |   5  |   4  |   3  |   2  |   1  |   0  |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| size  |  0    |  0    |  1    |  0    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  0    |  0    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  0    |  0    |  0    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  0    |  0    |  0    |
| Pg    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |
| Zm    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |
| Zdn   |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |  1    |  0    |
```

SQRSHL $<Zdn>$.<T>, $<Pg>/M$, $<Zdn>$.<T>, $<Zm>$.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assembler Symbols**

$<Zdn>$ Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

$<T>$ Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$<Pg>$ Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

$<Zm>$ Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    if Elem[mask, e, esize] == '1' then
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = SignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQRSHLR

Signed saturating rounding shift left reversed vectors (predicated).

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

SQRSHLR <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, <Zm>,<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[m];
bits(VL) operand2 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  integer element = SInt(Elem[operand1, e, esize]);
  integer shift = ShiftSat(Int(Elem[operand2, e, esize]), esize);
  integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
  if Elem[mask, e, esize] == '1' then
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];
  Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQRSHRNB

Signed saturating rounding shift right narrow by immediate (bottom).

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range \((-2^{(N-1)})\) to \((2^{(N-1)})\)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
0 1 0 0 0 1 0 1 0 1 0  tszh 1  tszl imm3 0 0 1 0 1 0 Zn  Zd

SQRSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>   Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const>  Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (SInt(element) + round_const) >> shift;
  Elem[result, 2*e + 0, esize] = SignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros();
Z[d] = result;
Signed saturating rounding shift right narrow by immediate (top).

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
0  1  0  0  0  1  0  1  0  1  1  0  0  1  0  1  1  0  1  0  1  0  1  0  1  0  1  0
0  1  0  0  1  0  1  0  1  1  0  0  1  0  1  1  0  0  1  0  1  0  1  0  1  0  1  0

SQRSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>
```

- `<Zd>` is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<const>` is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

```
if !(HaveSVE2()) then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsz:imm3);
```

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (SInt(element) + round_const) >> shift;
  Elem[result, 2*e + 1, esize] = SignedSat(res, esize);
Z[d] = result;
```
SQRSHRUNB

Signed saturating rounding shift right unsigned narrow by immediate (bottom).

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width $N$-bit element's unsigned integer range $0$ to $(2^N)-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

### Assembler Symbols

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “tszh:tszl”:
  ```
<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>
  ```
- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.
- **<Tb>** Is the size specifier, encoded in “tszh:tszl”:
  ```
<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>
  ```
- **<const>** Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

### Operation

```pseudo
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (SInt(element) + round_const) >> shift;
  Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros();

Z[d] = result;
```
SQRSHRUNT

Signed saturating rounding shift right unsigned narrow by immediate (top).

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element’s unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-------------|-------------|----------|
| 0 1 0 0 1 0 1 0 0 0 1 0 1 0 | tszh tszl   | imm3 0 0 0 |
| U R T                      | Zn Zd       |          |
```

SQRSHRUNT <Zd>.<T>, <Zn>.<Tb>, #<const>

if !HaveSVE2() then UNDEFINED;

bits(3) tsize = tszh:tszl;

case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;

integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Asmmeber Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();

integer elements = VL DIV (2 * esize);

bits(VL) operand = Z[n];

bits(VL) result = Z[d];

integer round_const = 1 << (shift-1);

for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (SInt(element) + round_const) >> shift;
  Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
SQSHL (immediate)

Signed saturating shift left by immediate.

Shift left by immediate each active signed element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range \( -2^{(N-1)} \) to \( 2^{(N-1)}-1 \). The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 1 0 0 0 1 1 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0

SQSHL <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, ##<const>
```

if !HaveSVEZ() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
temp integer shift = UInt(tsize:imm3) - esize;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = element1 << shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
  Elem[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQSHL (vectors)**

Signed saturating shift left by vector (predicated).

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Inactive elements in the destination vector register remain unmodified.

```
0 1 0 0 0 1 0 0 | size 0 0 1 0 0 0 0 0 | Pg 1 0 0 1 0 0 | Zm | Zdn
```

SQSHL <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, <Zm>..<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assemble Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the ”Zdn” field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element = SInt(Elem[operand1, e, esize]);
  integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
  if Elem[mask, e, esize] == '1' then
    integer res = element << shift;
    Elem[result, e, esize] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed saturating shift left reversed vectors (predicated).

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). Inactive elements in the destination vector register remain unmodified.

```
 0 1 0 0 0 0 1 0 0 | size | 0 0 1 1 0 0 0 1 0 0 | Pg | Zm | Zdn
```

if \(!\text{HaveSVE2}()\) then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer g = \text{UInt}(Pg);
integer m = \text{UInt}(Zm);
integer dn = \text{UInt}(Zdn);

**Assembler Symbols**

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```
CheckSVEEnabled();
integer elements = \text{VL DIV esize};
bits(PL) mask = \text{P}[g];
bits(VL) operand1 = \text{Z}[m];
bits(VL) operand2 = \text{Z}[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element = \text{SInt(Elem[operand1, e, esize])};
  integer shift = \text{ShiftSat(SInt(Elem[operand2, e, esize]), esize)};
  if \text{ElemP[mask, e, esize]} == '1' then
    integer res = element << shift;
    \text{Elem[result, e, esize]} = \text{SignedSat(res, esize)};
  else
    \text{Elem[result, e, esize]} = \text{Elem[operand2, e, esize]};
\text{Z}[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a \text{MOVPRFX} instruction. The \text{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \text{MOVPRFX} and this instruction is UNPREDICTABLE:

- The \text{MOVPRFX} instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQSHLU

Signed saturating shift left unsigned by immediate.

Shift left by immediate each active signed element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

SQSHLU <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

if !HaveSVE2() then UNDEFINED;
bites(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bites(VL) operand1 = Z[dn];
bites(PL) mask = P[g];
bites(VL) result;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = element1 << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQSHRNB

Signed saturating shift right narrow by immediate (bottom).

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)}-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 0 0 1 0 1 0 | tszh | tszl | imm3 | 0 0 1 0 0 0 | Zn | Zd |
| U | R | T |

**SQSHRNB** <Zd>.<T>, <Zn>.<Tb>, #<const>

```plaintext
if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsz:imm3);
```

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the “Zn” field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = SInt(element) >> shift;
  Elem[result, 2*e + 0, esize] = SignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros();
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
SQSHRNT

Signed saturating shift right narrow by immediate (top).

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element’s signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)}) - 1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

\[
\begin{array}{ccccccccccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & tszh & 1 & tszl & imm3 & 0 & 0 & 1 & 0 & 0 & 1 & Zn & Zd & U & R & T
\end{array}
\]

**SQSHRNT \(<Zd>\), \(<T>\), \(<Zn>\), \(<Tb>\), \#<const>**

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

**Assembler Symbols**

\(<Zd>\) Is the name of the destination scalable vector register, encoded in the "Zd" field.
\(<T>\) Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tzh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.
\(<Tb>\) Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tzh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = SInt(element) >> shift;
  Elem[result, 2*e + 1, esize] = SignedSat(res, esize);
Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
SQSHRUNB

Signed saturating shift right unsigned narrow by immediate (bottom).

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 1 0 tszh 1 tszl imm3 0 0 0 0 0 0 Zn Zd</td>
</tr>
</tbody>
</table>

SQSHRUNB <Zd>.<T>, <Zn>.<T>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols
<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>1 xx</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>1 xx</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = SInt(element) >> shift;
  Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros();
Z[d] = result;
Signed saturating shift right unsigned narrow by immediate (top).

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```plaintext
if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsz:imm3);
```

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “tszh:tszl”:
  
<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>
- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Tb>** Is the size specifier, encoded in “tszh:tszl”:
  
<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>
- **<const>** Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VLDIV (2 * esize);
bv(VL) operand = Z[n];
bv(VL) result = Z[d];
for e = 0 to elements-1
  bv(2*esize) element = Elem[operand, e, 2*esize];
  integer res = SInt(element) >> shift;
  Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);
Z[d] = result;
```
SQSUB (vectors, predicated)

Signed saturating subtraction (predicated).

Subtract active signed elements of the second source vector from corresponding signed elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Pg | Zm | Zdn |

SQSUB <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, <Zm>..<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<size> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = SInt(Sat(element1 - element2, esize, unsigned));
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SQSUB (immediate)**

Signed saturating subtract immediate (unpredicated).

Signed saturating subtract of an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

**Assembler Symbols**

\(<\text{Zdn}>\) Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

\(<\text{T}>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{imm}>\) Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

\(<\text{shift}>\) Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SQSUB (vectors, unpredicated)

Signed saturating subtract vectors (unpredicated).

Signed saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element’s signed integer range \(-2^{(N-1)}\) to \(2^{(N-1)}-1\). This instruction is unpredicated.

```
0 0 0 0 0 1 0 0  | size | 1  | Zm | 0 0 0 1 1 0 | Zn | Zd
```

SQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);
Z[d] = result;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQSUBR

Signed saturating subtraction reversed vectors (predicated).

Subtract active signed elements of the first source vector from corresponding signed elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range \(-2^{(N-1)}\) to \((2^{(N-1)})-1\). Inactive elements in the destination vector register remain unmodified.

```plaintext
0 1 0 0 0 1 1 1 0 1 0 0 0 1 0 0
Pg Zm Zdn
```

SQSUBR \(<\text{Zdn}>.\text{<T>}, <\text{Pg}>/M, <\text{Zdn}>.\text{<T>}, <\text{Zm}>.\text{<T>}

if !\text{HaveSVE2}() then UNDEFINED;
integer esize = 8 << \text{UInt} \text{(size)};
integer g = \text{UInt} \text{(Pg)};
integer dn = \text{UInt} \text{(Zdn)};
integer m = \text{UInt} \text{(Zm)};
boolean unsigned = FALSE;

### Assembler Symbols

\(<\text{Zdn}>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<\text{T}>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<\text{Zm}>\) Is the name of the second source scalable vector register, encoded in the “Zm” field.

### Operation

\text{CheckSVEEnabled}();
integer elements = \text{VL} \text{ DIV} esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = \text{SInt(Elem[operand1, e, esize]);}
    integer element2 = \text{SInt(Elem[operand2, e, esize]);}
    if Elem[mask, e, esize] == '1' then
        integer res = \text{SInt(Sat(element2 - element1, esize, unsigned));}
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Signed saturating extract narrow (bottom).

Saturate the signed integer value in each source element to half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 1 0 1 0 | tszh | 1 tszl | 0 0 0 0 1 0 0 | 0 | Zn | | Zd
```

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Zn> is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
integer halfesize = esize DIV 2;
for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
bits(halfesize) res = SignedSat(element1, halfesize);
    Elem[result, 2*e + 0, halfesize] = res;
    Elem[result, 2*e + 1, halfesize] = Zeros();
Z[d] = result;
Signed saturating extract narrow (top).

Saturate the signed integer value in each source element to half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 0 0 1 0 1 0 tszh | 1 tszl | 0 0 0 1 0 0 0 1 | Zn | Zd |

SQXTNT <Zd>,<T>, <Zn>,<Tb>

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '001' esize = 16;
  when '010' esize = 32;
  when '100' esize = 64;
  otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>  Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>x</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb>  Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>x</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result = Z[d];
integer halfesize = esize DIV 2;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
bits(halfesize) res = SignedSat(element1, halfesize);
  Elem[result, 2*e + 1, halfesize] = res;
Z[d] = result;
SQXTUNB

Signed saturating unsigned extract narrow (bottom).

Saturate the signed integer value in each source element to an unsigned integer value that is half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 0   | 0   | 1   | 1   | tszh | 1   | tszl | 0   | 0   | 0   | 1   | 0   | 1   | 0   | 1   | Zn  | Zd  |

if !HaveSVE2() then UNDEFINED;

bits(3) tsize = tszh:tszl;
case tsize of
when '001' esize = 16;
when '010' esize = 32;
when '100' esize = 64;
otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
integer halfesize = esize DIV 2;
for e = 0 to elements-1
   integer element1 = SInt(Elem[operand1, e, esize]);
   bits(halfesize) res = UnsignedSat(element1, halfesize);
   Elem[result, 2*e + 0, halfesize] = res;
   Elem[result, 2*e + 1, halfesize] = Zeros();
Z[d] = result;
Signed saturating unsigned extract narrow (top).

Saturate the signed integer value in each source element to an unsigned integer value that is half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>D</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

for e = 0 to elements-1
   integer element1 = SIInt(Elem[operand1, e, esize]);
   bits(halfesize) res = UnsignedSat(element1, halfesize);
   Elem[result, 2*e + 1, halfesize] = res;
Z[d] = result;
SRHADD

Signed rounding halving addition.

Add active signed elements of the first source vector to corresponding signed elements of the second source vector, shift right one bit, and destructively place the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | Pg | Zm | Zdn |

SRHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer round_const = 1;
for e = 0 to elements-1
  integer element1 = SInt(Elem[operand1, e, esize]);
  integer element2 = SInt(Elem[operand2, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = element1 + element2 + round_const;
    Elem[result, e, esize] = res<esize:1>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SRI

Shift right and insert (immediate).

Shift each source vector element right by an immediate value, and insert the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
0 1 0 0 1 0 1 tzh 0 tzl imm3 1 1 1 0 0 Zn  Zd
```

SRI <Zd>.<T>, <Zn>.<T>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tzh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
  bits(esize) element1 = Elem[result, e, esize];
  bits(esize) element2 = Elem[operand, e, esize];
  bits(esize) mask = LSR(Ones(esize), shift);
  bits(esize) shiftedval = LSR(element2, shift);
  Elem[result, e, esize] = (element1 AND (NOT mask)) OR shiftedval;

Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
Signed rounding shift left by vector (predicated).

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
```

SRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    if Elem[mask, e, esize] == '1' then
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SRSHLR

Signed rounding shift left reversed vectors (predicated).

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 1 0 0</td>
<td>0 0</td>
<td>1 1 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[m];
bits(VL) operand2 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    if Elem[mask, e, esize] == '1' then
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];
    Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SRSHR

Signed rounding shift right by immediate.

Shift right by immediate each active signed element of the source vector, and destructively place the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 1 0 0 | tszh | 0 0 | 1 1 | 0 0 | 1 0 0 | Pg | tszl | imm3 | Zdn |

SRSHR <Zdn><T>, <Pg>/M, <Zdn><T>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zdn>        Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T>          Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>00 01</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>00 1x</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>01 xx</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>1x xx</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

<Pg>            Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<const>         Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
integer round Const = 1 << (shift-1);
for e = 0 to elements-1
integer element1 = SInt(Elem[operand1, e, esize]);
if ElemP[mask, e, esize] == '1' then
    integer res = (element1 + round Const) >> shift;
    Elem[result, e, esize] = res<esize-1:0>;
else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SRSRA

Signed rounding shift right and accumulate (immediate).

Shift right by immediate each signed element of the source vector, preserving the sign bit, and add the rounded intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | tzh | 0  | tszl | imm3 | 1  | 1  | 1  | 0  | 1  | 0  | Zn  | Zda |

SRSRA <Zda>,<T>, <Zn>,<T>, #<const>

if (!HaveSVE2()) then UNDEFINED;
bits(4) tsize = tzh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;

integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsz:imm3);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tzh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVEd</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[da];
bits(VL) result;
integer round_const = 1 << (shift - 1);

for e = 0 to elements-1
  integer element = (SInt(Elem[operand1, e, esize]) + round_const) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SSHLLB

Signed shift left long by immediate (bottom).

Shift left by immediate each even-numbered signed element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | U | T |

SSHLLB <Zd>,<T>, <Zn>,<Tb>, #<const>

if !HaveSVEz() then UNDEFINED;
bhits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<T> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Tb> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bhits(VL) operand = Z[n];
bhits(VL) result;
for e = 0 to elements-1
  bits(esize) element = Elem[operand, 2*e + 0, esize];
  integer shifted_value = SInt(element) << shift;
  Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;
Z[d] = result;
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SSHLLT**

Signed shift left long by immediate (top).

Shift left by immediate each odd-numbered signed element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

|   | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|   | tszh | tszl | imm3 | Zn | Zd |

SSHLLT `<Zd>,<T>, <Zn>,<Tb>, #<const>

if ![HaveSVE2()] then UNDEFINED;

bits(3) tsize = tszh:tszl;

```plaintext
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>1x</td>
<td>01</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<const>` Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tszh:imm3”.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element = Elem[operand, 2*e + 1, esize];
  integer shifted_value = SInt(element) << shift;
  Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;
Z[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed shift right and accumulate (immediate).

Shift right by immediate each signed element of the source vector, preserving the sign bit, and add the truncated intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
  31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
  0  1  0  0  0  1  0  1  tszh  0  tszl  imm3  1  1  1  0  0  0  Zn  Zda
```

#### SSRA <Zda>.<T>, <Zn>.<T>, #<const>

if !HaveSVEz() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UINT(Zn);
integer da = UINT(Zda);
integer shift = (2 * esize) - UINT(tsize:imm3);

#### Assembler Symbols

- `<Zda>`: Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>`: Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>`: Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<const>`: Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

#### Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  integer element = SInt(Elem[operand1, e, esize]) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
Z[da] = result;
```

#### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
The values of the NZCV flags.

This instruction might be immediately preceded in program order by a **MOVPRFX** instruction. The **MOVPRFX** instruction must conform to all of the following requirements, otherwise the behavior of the **MOVPRFX** and this instruction is **UNPREDICTABLE**:

- The **MOVPRFX** instruction must be unpredicated.
- The **MOVPRFX** instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SSUBLB

Signed subtract long (bottom).

Subtract the even-numbered signed elements of the second source vector from the corresponding signed elements of
the first source vector, and place the results in the overlapping double-width elements of the destination vector. This
instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | Zm | 0  | 0  | 1  | 0  | 0  | Zn | Zd |

SSUBLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = FALSE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSUBLBT

Signed subtract long (bottom - top).

Subtract the odd-numbered signed elements of the second source vector from the even-numbered signed elements of
the first source vector, and place the results in the overlapping double-width elements of the destination vector. This
instruction is unpredicated.

SSUBLBT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
boolean unsigned = FALSE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSUBLT

Signed subtract long (top).

Subtract the odd-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 | size | 0 | Zm | 0 0 1 0 1 | Zn | Zd
        S  U  T
```

SSUBLT <Zd>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = FALSE;

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the “Zd” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed subtract long (top - bottom).

Subtract the even-numbered signed elements of the second source vector from the odd-numbered signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

SSUBLTB

SSUBLTB \(<Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>\)

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 0;
boolean unsigned = FALSE;

Assembler Symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>   Is the size specifier, encoded in “size”:

\[
\begin{array}{c|c}
\text{size} & <T> \\
\hline
00 & \text{RESERVED} \\
01 & H \\
10 & S \\
11 & D \\
\end{array}
\]

<Zn>  Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb>  Is the size specifier, encoded in “size”:

\[
\begin{array}{c|c}
\text{size} & <Tb> \\
\hline
00 & \text{RESERVED} \\
01 & B \\
10 & H \\
11 & S \\
\end{array}
\]

<Zm>  Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = \(\text{VL} \div \text{esize}\);
bits(\text{VL}) operand1 = \(\text{Z}[n]\);
bits(\text{VL}) operand2 = \(\text{Z}[m]\);
bits(\text{VL}) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
  integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
  integer res = element1 - element2;
  Elem[result, e, esize] = res<esize-1:0>;
\Z[d] = result;
Operational information

If \text{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SSUBWB

Signed subtract wide (bottom).

Subtract the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 0 & Zm & 0 & 1 & 0 & 1 & 0 & 0 & \text{Zn} & \text{Zd} \\
\end{array}
\]

SSUBWB \text{<Zd>.<T>}, \text{<Zn>.<T>}, \text{<Zm>.<Tb>}

if \text{!HaveSVE2() then UNDEFINED;}
if size == '00' then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer n = \text{UInt}(Zn);
integer m = \text{UInt}(Zm);
integer d = \text{UInt}(Zd);

Assembler Symbols

\text{<Zd>} Is the name of the destination scalable vector register, encoded in the "Zd" field.
\text{<T>} Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\text{<Zn>} Is the name of the first source scalable vector register, encoded in the "Zn" field.
\text{<Zm>} Is the name of the second source scalable vector register, encoded in the "Zm" field.
\text{<Tb>} Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

Operation

\text{CheckSVEEnabled();}
integer elements = \text{VL} \text{DIV} esize;
bits(\text{VL}) operand1 = \text{Z}[n];
bits(\text{VL}) operand2 = \text{Z}[m];
bits(\text{VL}) result;
for e = 0 to elements-1
  integer element1 = \text{SInt}(\text{Elem}[operand1, e, esize]);
  integer element2 = \text{SInt}(\text{Elem}[operand2, 2*e + 0, esize \text{DIV} 2]);
  \text{Elem}[result, e, esize] = (element1 - element2)<\text{esize}-1:0>;
\text{Z}[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**SSUBWT**

Signed subtract wide (top).

Subtract the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>0 1 0 1 0 1 0 1</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**SSUBWT** <Zd>.,<T>, <Zn>.<T>, <Zm>..<Tb>

if !HaveSVE2() then UNDEFINED;
if size == ‘00’ then UNDEFINED;
in integer esize = 8 << UInt(size);
in integer n = UInt(Zn);
in integer m = UInt(Zm);
in integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the “Zd” field.
- `<T>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.
- `<Tb>` Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

**Operation**

- `CheckSVEEnabled();`
- `integer elements = VL DIV esize;`
- `bits(VL) operand1 = Z[n];`
- `bits(VL) operand2 = Z[m];`
- `bits(VL) result;`
- `for e = 0 to elements-1`
  - `integer element1 = SInt(Elem[operand1, e, esize]);`
  - `integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);`
  - `Elem[result, e, esize] = (element1 - element2)<esize-1:0>;`
- `Z[d] = result;`

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
ST1B (vector plus immediate)

Scatter store bytes from a vector (immediate index).

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements are not written to memory.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | imm5| 1  | 0  | 1  | Pg | Zn | Zt |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

ST1B { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>imm5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ST1B { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
**ST1B (scalar plus immediate)**

Contiguous store bytes from vector (immediate index).

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0 0</td>
</tr>
</tbody>
</table>

msz<1>msz<0>

ST1B { <Zt>,<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 8;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
  addr = addr + mbytes;
ST1B (scalar plus scalar)

Contiguous store bytes from vector (scalar index).

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<table>
<thead>
<tr>
<th>size</th>
<th>Rm</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ST1B { <Zt>,<T>, <Pg>, [Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 8;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
  offset = offset + 1;

ST1B (scalar plus vector)

Scatter store bytes from a vector (vector index).

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements are not written to memory.

It has encodings from 3 classes: 32-bit unpacked unscaled offset, 32-bit unscaled offset and 64-bit unscaled offset

32-bit unpacked unscaled offset

\[
\begin{array}{cccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & Zm & 1 & xs & 0 & Pg & Rn & Zt
\end{array}
\]

\[
\text{msz}<1>\text{msz}<0>
\]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

\[
\begin{array}{cccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & Zm & 1 & xs & 0 & Pg & Rn & Zt
\end{array}
\]

\[
\text{msz}<1>\text{msz}<0>
\]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit unscaled offset

\[
\begin{array}{cccccccccccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & Zm & 1 & 0 & 1 & Pg & Rn & Zt
\end{array}
\]

\[
\text{msz}<1>\text{msz}<0>
\]
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm>  Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(VL) offset = Z[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSAlignment();
  base = SP[];
else
  base = X[n];

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
ST1D (vector plus immediate)

Scatter store doublewords from a vector (immediate index).

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements are not written to memory.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 1 1 0 0 1 0 | 1 | 1 | 1 0 | imm5 | 1 0 1 | Pg | Zn | Zt |

msz<1>msz<0>

ST1D { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if Elem[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST1D (scalar plus immediate)

Contiguous store doublewords from vector (immediate index).

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|----------------|-----------------|-----------------|-----------------|-----------------|
| 1 1 1 0 0 1 0 1 | 1 1 1 size 0 imm4 | 1 1 1 Pg | Rn | Zt |

msz<1>msz<0>

ST1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 64;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPEAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
    addr = addr + mbytes;
ST1D (scalar plus scalar)

Contiguous store doublewords from vector (scalar index).

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | Rm | 0  | 1  | 0  | Pg | Rn | Zt |

ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
   if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   base = SP[];
else
   base = X[n];
for e = 0 to elements-1
   addr = base + UInt(offset) * mbytes;
   if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
   offset = offset + 1;
**ST1D (scalar plus vector)**

Scatter store doublewords from a vector (vector index).

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements are not written to memory.

It has encodings from 4 classes: 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

### 32-bit unpacked scaled offset

```
0 1 1 1 0 0 0 1 1 1 0 1 Zm 1 xs 0 Pg Rn Zt
```

```
ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 3;

### 32-bit unpacked unscaled offset

```
0 1 1 1 0 0 0 1 1 1 0 0 Zm 1 xs 0 Pg Rn Zt
```

```
ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

### 64-bit scaled offset

```
0 1 1 1 0 0 0 1 1 1 0 1 Zm 1 0 1 Pg Rn Zt
```

ST1D (scalar plus vector)
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 64;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | Zm | 1  | 0  | 1  |Pg | Rn | Zt |

msz<1>msz<0>

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(VL) offset = Z[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
```

ST1H (vector plus immediate)

Scatter store halfwords from a vector (immediate index).

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements are not written to memory.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ST1H { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]  

if !HaveSVE() then UNDEFINED;  
integer t = UInt(Zt);  
integer n = UInt(Zn);  
integer g = UInt(Pg);  
integer esize = 32;  
integer msize = 16;  
integer offset = UInt(imm5);

64-bit element

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>imm5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ST1H { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]  

if !HaveSVE() then UNDEFINED;  
integer t = UInt(Zt);  
integer n = UInt(Zn);  
integer g = UInt(Pg);  
integer esize = 64;  
integer msize = 16;  
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

Operation

`CheckSVEEnabled();`

```
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
```

```
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
```

```
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
```
**ST1H (scalar plus immediate)**

Contiguous store halfwords from vector (immediate index).

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
**ST1H (scalar plus scalar)**

Contiguous store halfwords from vector (scalar index).

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  |
```

ST1H { <Zt>,<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 16;

**Assembler Symbols**

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<T>`: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = Xm[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
        base = SP[];
    else
        base = Xm[n];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
    offset = offset + 1;
```
ST1H (scalar plus vector)

Scatter store halfwords from a vector (vector index).

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements are not written to memory.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset

32-bit scaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

msz<1>msz<0>

ST1H { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked scaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

msz<1>msz<0>

ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked unscaled offset

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

64-bit scaled offset

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 1;

64-bit unscaled offset
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod> Is the index extend and shift specifier, encoded in “xs”:

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(VL) offset = Z[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckS PLAignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
ST1W (vector plus immediate)

Scatter store words from a vector (immediate index).

Scatter store of words from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements are not written to memory.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

```
+---+---+---+---+---+---+---+---+---+---+---+---+
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
+---+---+---+---+---+---+---+---+---+---+---+---|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---|
| imm5|   |   |   | Pg |   |   |   |   | Zn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---|
| msz<1> msz<0> |
```

ST1W { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offset = UInt(imm5);

64-bit element

```
+---+---+---+---+---+---+---+---+---+---+---+---+
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
+---+---+---+---+---+---+---+---+---+---+---+---|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---|
| imm5|   |   |   | Pg |   |   |   |   | Zn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+---|
| msz<1> msz<0> |
```

ST1W { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offset = UInt(imm5);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.
Operation

*CheckSVEEnabled*();

integer elements = *VL* DIV *esize*;

bits(*VL*) base = *Z*[n];

bits(*VL*) src = *Z*[t];

bits(*PL*) mask = *P*[g];

bits(64) addr;

constant integer mbytes = msize DIV 8;

if *HaveMTEExt*() then *SetTagCheckedInstruction*(TRUE);  

for e = 0 to elements-1
   if *ElemP*[mask, e, esize] == '1' then
      addr = *ZeroExtend*(*Elem*[base, e, esize], 64) + offset * mbytes;
      *Mem*[addr, mbytes, AccType_NORMAL] = *Elem*[src, e, esize]<msize-1:0>;
ST1W (scalar plus immediate)

Contiguous store words from vector (immediate index).

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 0 0 0 1 | size | imm4 | 1 1 1 | Pg | Rn | Zt

msz<1>msz<0>

ST1W { <Zt>.<T> }, <Pg>, [<Xn|SP>({, #<imm>, MUL VL})

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 32;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;

if n == 31 then
    if LastActiveElement(mask, esize) == 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
else
    addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
    addr = addr + mbytes;
ST1W (scalar plus scalar)

Contiguous store words from vector (scalar index).

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |   |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|   |
|   | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |   |    |    |    |    |    |    |    |    |    |    |    |    |    |   |

ST1W { <Zt>,<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<T> Is the size specifier, encoded in "size<0>":

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[1];
else
    base = X[n];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP(mask, e, esize) == '1' then
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
    offset = offset + 1;
ST1W (scalar plus vector)

Scatter store words from a vector (vector index).

Scatter store of words from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements are not written to memory.

It has encodings from 6 classes: 32-bit scaled offset, 32-bit unpacked scaled offset, 32-bit unpacked unscaled offset, 32-bit unscaled offset, 64-bit scaled offset and 64-bit unscaled offset.

**32-bit scaled offset**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 0 1 1</td>
</tr>
</tbody>
</table>
```

msz<1>msz<0>

ST1W { <Zt> . S }, <Pg>, [<Xn|SP>], <Zm>. S, <mod> #2

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

**32-bit unpacked scaled offset**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 0 1 1</td>
</tr>
</tbody>
</table>
```

msz<1>msz<0>

ST1W { <Zt> . D }, <Pg>, [<Xn|SP>], <Zm>. D, <mod> #2

```c
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

**32-bit unpacked unscaled offset**

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 0 0</td>
</tr>
</tbody>
</table>
```

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer t = UINT(Zt);
integer n = UINT(Rn);
integer m = UINT(Zm);
integer g = UINT(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0 | Zm | 1 | xs | 0 | Pg | Rn | Zt |

msz<1>msz<0>

64-bit scaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1 | Zm | 1 | 0 | 1 | Pg | Rn | Zt |

msz<1>msz<0>

64-bit unscaled offset

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0 | Zm | 1 | 0 | 1 | Pg | Rn | Zt |

msz<1>msz<0>
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;

Assembler Symbols

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm>  Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod>  Is the index extend and shift specifier, encoded in "xs":

<table>
<thead>
<tr>
<th>xs</th>
<th>&lt;mod&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UXTW</td>
</tr>
<tr>
<td>1</td>
<td>SXTW</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(VL) offset = Z[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictable Bool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
    addr = base + (off << scale);
    Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;
ST2B (scalar plus immediate)

Contiguous store two-byte structures from two vectors (immediate index).

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector’s in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

**Assembler Symbols**

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
  else if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

  for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

  addr = base + offset * elements * nreg * mbytes;
  for e = 0 to elements-1
    for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
```


ST2B (scalar plus scalar)

Contiguous store two-byte structures from two vectors (scalar index).

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 0 0 1 Rm 0 1 1 Pg Rn Zt
msz<1>msz<0>
```

ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 2;
```

Assembler Symbols

- `<Zt1>` is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
  else
    base = SP[ ];
else
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
  addr = addr + mbytes;
  offset = offset + nreg;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST2D (scalar plus immediate)

Contiguous store two-doubleword structures from two vectors (immediate index).

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 1 1 0 1 0 1 0 1 1 imm4 1 1 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

msz<1> msz<0>

ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>{{, #<imm>, MUL VL}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 2;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
        addr = addr + mbytes;

ST2D (scalar plus immediate)
STS2D (scalar plus scalar)

Contiguous store two doubleword structures from two vectors (scalar index).

Contiguous store two doubleword structures, each from the same element number in two vector registers to
the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL
option) and added to the base address. After each structure access the index value is incremented by two. The index
register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the
two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.

<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
     ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
     CheckSPAlignment();
else
  base = SP[];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST2H (scalar plus immediate)

Contiguous store two-halfword structures from two vectors (immediate index).

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector’s in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

ST2H (scalar plus immediate)

\[
\begin{array}{cccccccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & \text{imm4} & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

\text{msz<1>msz<0>}

\[
\text{ST2H} \{ <Zt1>.H, <Zt2>.H }, <\text{Pg}>, [<\text{Xn}|\text{SP}>\{, #<\text{imm}>, \text{MUL VL}]}\]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 2;

Assembler Symbols

\(<Zt1>\) Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
\(<Zt2>\) Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
\(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\(<\text{Xn}|\text{SP}>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\(<\text{imm}>\) Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

\text{CheckSVEEnabled}();
integer elements = \text{VL DIV esize};
bits(64) base;
bits(64) addr;
bits(\text{PL}) mask = \text{P}[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(\text{VL}) values;

if n == 31 then
  if \text{LastActiveElement}(mask, esize) >= 0 ||
    \text{ConstrainUnpredictableBool}(\text{Unpredictable_CheckSPNoneActive}) then
      \text{CheckSPAlignment}();
  if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(FALSE);
  base = \text{SP}[];
else
  if \text{HaveMTEExt}() then \text{SetTagCheckedInstruction}(TRUE);
  base = \text{X}[n];

for r = 0 to nreg-1
  values[r] = \text{Z}[(t+r) \text{ MOD } 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if \text{ElemP}[mask, e, esize] == '1' then
      \text{Mem}[addr, mbytes, \text{AccType_NORMAL}] = \text{Elem}[values[r], e, esize];
    addr = addr + mbytes;
ST2H (scalar plus scalar)

Contiguous store two-halfword structures from two vectors (scalar index).

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  |
```

msz<1> msz<0>

ST2H { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 2;

Assembler Symbols

<Zt> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ELEM[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
  offset = offset + nreg;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST2W (scalar plus immediate)

Contiguous store two-word structures from two vectors (immediate index).

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.

```assembly
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1
msz<1>msz<0>

ST2W { <Zt1>.S, <Zt2>.S }, <Pg>, [ <Xn|SP>{, #<imm>, MUL VL} ]
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 2;

### Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

### Operation

```assembly
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrantUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
  else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType.NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
```

ST2W (scalar plus immediate)
**ST2W (scalar plus scalar)**

Contiguous store two-word structures from two vectors (scalar index).

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 0 0 1 0 1 0 1 0 1 1 0 1 1 0 1 0
msz<1> msz<0>
```

```
ST2W \{ <Zt1>.S, <Zt2>.S \}, <Pg>, \{<Xn|SP>, <Xm>, LSL #2\}
```

```
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 2;
```

**Assembler Symbols**

- `<Zt>` is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

`CheckSVEEnabled();`
integer elements = \( VL \) DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = \( P \)[g];
bits(64) offset = \( X \)[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(\( VL \)) values;

if \( \text{HaveMTEExt}() \) then \( \text{SetTagCheckedInstruction}(\text{TRUE}); \)

if n == 31 then
  if \( \text{LastActiveElement}(\text{mask, esize}) >= 0 \) ||
    \( \text{ConstrainUnpredictableBool}((\text{Unpredictable}_\text{CHECKSPNONEACTIVE}) \) then
    \( \text{CheckSPAlignment}(); \)
  else
    base = \( SP \)[ ];
else
  base = \( X \)[n];

for r = 0 to nreg-1
  values[r] = \( Z \)[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if \( \text{ElemP}[\text{mask, e, esize}] == '1' \) then
      \( \text{Mem}[\text{addr, mbytes, AccType.NORMAL}] = \text{Elem}[\text{values}[r], e, esize]; \)
    addr = addr + mbytes;
    offset = offset + nreg;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3B (scalar plus immediate)

Contiguous store three-byte structures from three vectors (immediate index).

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```
31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
1  1  1  0  0  1  0  0  1  0  1  imm4  1  1  Pg  Rn  Zt
```


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAloignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
        addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3B (scalar plus scalar)

Contiguous store three-byte structures from three vectors (scalar index).

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

```assembly
ST3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>, [<Xn|SP>, <Xm>]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 3;

Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
    offset = offset + nreg;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3D (scalar plus immediate)

Contiguous store three-doubleword structures from three vectors (immediate index).

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 | 1 | 1 1 0 1 | imm4 | 1 1 1 | Pg | Rn | Zt
```

msz<1> msz<0>

ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = Uint(Zt);
integer n = Uint(Rn);
integer g = Uint(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3D (scalar plus scalar)

Contiguous store three-doubleword structures from three vectors (scalar index).

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```
ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

```plaintext
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer nreg = 3;
```

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  base = SP[];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3H (scalar plus immediate)

Contiguous store three-halfword structures from three vectors (immediate index).

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | imm4 | 1 | 1 | 1 | Pg | Rn | Zt |

msz<1>msz<0>


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
        addr = addr + mbytes;
```
ST3H (scalar plus scalar)

Contiguous store three-halfword structures from three vectors (scalar index).

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

```
1 1 0 0 1 0 1 0 Rm 0 1 1 Pg Rn Zt
```

ST3H \{ <Zt1>.H, <Zt2>.H, <Zt3>.H \}, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;
integer nreg = 3;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  base = SP[];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**ST3W (scalar plus immediate)**

Contiguous store three-word structures from three vectors (immediate index).

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 0 1 0 1 imm4 1 1 1 Pg Rn Zt
msz<1>msz<0>
```

ST3W \{ <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```java
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 3;
```

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the “imm4” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
  else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST3W (scalar plus scalar)

Contiguous store three-word structures from three vectors (scalar index).

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.

ST3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer nreg = 3;

Assembler Symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>  Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;
if HaveMTEEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrunpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
else
  base = SP[];
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
ST4B (scalar plus immediate)

Contiguous store four-byte structures from four vectors (immediate index).

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0 0 1 0</td>
</tr>
</tbody>
</table>


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn\|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PC) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
        addr = addr + mbytes;
**ST4B (scalar plus scalar)**

Contiguous store four-byte structures from four vectors (scalar index).

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction. Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| msz<1>          | msz<0>          | Rm              | Pg              | Rn              | Zt              |


if !HaveSVE() then UNDEFINED;
if Rm == '1111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;
integer nreg = 4;

**Assembler Symbols**

- **<Zt1>** Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- **<Zt2>** Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- **<Zt3>** Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- **<Zt4>** Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Xn|SP>** Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- **<Xm>** Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPA1ignment();
  else
    base = X[n];
else
  base = SP[];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
      addr = addr + mbytes;
      offset = offset + nreg;
**ST4D (scalar plus immediate)**

Contiguous store four-doubleword structures from four vectors (immediate index).

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 1 1 1 1 1 1 1 1 imm4 1 1 1 Pg Rn Zt
msz<1>msz<0>
```

```
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer offset = SInt(imm4);
integer nreg = 4;

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
ST4D (scalar plus scalar)

Contiguous store four-doubleword structures from four vectors (scalar index).

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
\| 1 1 1 0 0 1 0 \| 1 \| 1 \| 1 0 1 1 \| 0 1 1 \| Pg \| Rn \| Zt
```

```
ST4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]
```

```
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer nreg = 4;
```

**Assembler Symbols**

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
   if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
   else
      base = SP[];
else
   base = X[n];

for r = 0 to nreg-1
   values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
   addr = base + UInt(offset) * mbytes;
   for r = 0 to nreg-1
      if ElemP[mask, e, esize] == '1' then
         Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
         addr = addr + mbytes;
         offset = offset + nreg;

Contiguous store four-halfword structures from four vectors (immediate index).

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory
address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that
is multiplied by the vector’s in-memory size, irrespective of predication.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the
four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

\[
\begin{array}{cccccccccccccccc}
\hline
1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & \text{imm4} & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]


if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
integer nreg = 4;

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0,
encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ST4H (scalar plus scalar)

Contiguous store four-halfword structures from four vectors (scalar index).

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.

Assembler Symbols

<Zt1>  Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>  Is the name of the second scalable vector register to be transferred, encoded as “Zt” plus 1 modulo 32.
<Zt3>  Is the name of the third scalable vector register to be transferred, encoded as “Zt” plus 2 modulo 32.
<Zt4>  Is the name of the fourth scalable vector register to be transferred, encoded as “Zt” plus 3 modulo 32.
<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>  Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>  Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
  if LastActiveElement(mask, esize) == 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];

for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];

for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
    offset = offset + nreg;
ST4W (scalar plus immediate)

Contiguous store four-word structures from four vectors (immediate index).

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 1 0 1 1 1 1 imm4 1 1 1 Pg Rn Zt

```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);
integer nreg = 4;

### Assembler Symbols

- `<Zt1>` Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
- `<Zt2>` Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
- `<Zt3>` Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
- `<Zt4>` Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>` Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits([PL]) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
        addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Contiguous store four-word structures from four vectors (scalar).

Contiguous store four-word structures, each from the same element number in four vector registers to the memory
address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option)
and added to the base address. After each structure access the index value is incremented by four. The index register
is not updated by the instruction.

Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the
four consecutive words in memory which make up each structure. Inactive structures are not written to memory.

Assembler Symbols

<Zt1> Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2> Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3> Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4> Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.
Operation

`CheckSVEEnabled();`

```plaintext
integer elements = VL DIV esize;
```

```
bits(64) base;
```  

```
bits(64) addr;
```  

```
bits(PL) mask = P[g];
```  

```
bites(64) offset = X[m];
```  

```
constant integer mbytes = esize DIV 8;
```  

```
array [0..3] of bits(VL) values;
```  

```
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
```

```
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  else
    base = SP[];
else
  base = X[n];
```

```
for r = 0 to nreg-1
  values[r] = Z[(t+r) MOD 32];
```

```
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  for r = 0 to nreg-1
    if ElemP[mask, e, esize] == '1' then
      Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
    addr = addr + mbytes;
    offset = offset + nreg;
```
STNT1B (vector plus scalar)

Scatter store non-temporal bytes.

Scatter store non-temporal of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

32-bit unscaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Rm  | 0  | 0  | 1  | Pg | Zn | Zt |

msz<1>msz<0>
```

```
STNT1B { <Zt>.S }, <Pg>, [<Zn>.S{}, <Xm>}

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 8;
```

64-bit unscaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1   | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Rm  | 0  | 0  | 1  | Pg | Zn | Zt |

msz<1>msz<0>
```

```
STNT1B { <Zt>.D }, <Pg>, [<Zn>.D{}, <Xm>}

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1B (scalar plus immediate)

Contiguous store non-temporal bytes from vector (immediate index).

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
            31  30  29  28  27  26  25  24  23  22  21  20  19  18  17  16  15  14  13  12  11  10  9  8  7  6  5  4  3  2  1  0
            +-----------------------------------------------------------+  
            | 1 1 1 0 0 1 0 0 0 1  | 0 0 0 1  | imm4 | 1 1 1 | Pg | Rn | Zt |
            +-----------------------------------------------------------+  
```

STNT1B { <Zt>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];

if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
        if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
        base = SP[];
    else
        if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
        base = X[n];
    src = Z[t];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
    addr = addr + mbytes;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1B (scalar plus scalar)

Contiguous store non-temporal bytes from vector (scalar index).

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------|---------------------------------------|
| msz<1> | msz<0> |

STNT1B { <Zt>.B }, <Pg>, [<Xn|SP>, <Xm>]
```

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 8;

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>`: Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset = X[m];
bits(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  base = SP[];
else
  base = X[n];
src = Z[t];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
  offset = offset + 1;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1D (vector plus scalar)

Scatter store non-temporal doublewords.

Scatter store non-temporal of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if !HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize]<msize-1:0>;
STNT1D (scalar plus immediate)

Contiguous store non-temporal doublewords from vector (immediate index).

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];
src = Z[t];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
  addr = addr + mbytes;
Contiguous store non-temporal doublewords from vector (scalar index).

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset = X[m];
bits(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
   if LastActiveElement(mask, esize) >= 0 ||
      ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
      base = SP[];
   else
      base = X[n];
   src = Z[t];
   for e = 0 to elements - 1
      addr = base + UInt(offset) * mbytes;
      if ElemP[mask, e, esize] == '1' then
         Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
      offset = offset + 1;
STNT1H (vector plus scalar)

Scatter store non-temporal halfwords.

Scatter store non-temporal of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

### 32-bit unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1><msz<0>

STNT1H {<Zt>.S},<Pg>, [<Zn>.S{,<Xm}>]

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 16;

### 64-bit unscaled offset

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

msz<1><msz<0>

STNT1H {<Zt>.D},<Pg>, [<Zn>.D{,<Xm}>]

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 16;

**Assembler Symbols**

<Zt>  Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg>  Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>  Is the name of the base scalable vector register, encoded in the "Zn" field.

<Xm>  Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

CheckSVEEnabled();
integer elements = $VL$ DIV $esize$;
bits($VL$) base = $Z\{n\}$;
bits(64) offset = $X\{m\}$;
bits($VL$) src = $Z\{t\}$;
bits($PL$) mask = $P\{g\}$;
bits(64) addr;
constant integer mbytes = $msize$ DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

for e = 0 to elements-1
  if $ElemP\{mask, e, esize\} == '1' then
    addr = ZeroExtend($Elem$[base, e, esize], 64) + offset;
    Mem[addr, mbytes, AccType_STREAM] = $Elem$[src, e, esize]$<msize-1:0>$;
STNT1H (scalar plus immediate)

Contiguous store non-temporal halfwords from vector (immediate index).

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

STNT1H { <Zt>, H }, <Pg>, [ <Xn|SP>, {, #imm}, MUL VL ]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
src = Z[t];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
  addr = addr + mbytes;
STNT1H (scalar plus scalar)

Contiguous store non-temporal halfwords from vector (scalar index).

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25</th>
<th>24 23 22 21 20 19 18</th>
<th>17 16 15 14 13 12 11 10</th>
<th>9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0 0</td>
<td>1 0 0</td>
<td>Rm 1 1 Pg</td>
<td></td>
</tr>
</tbody>
</table>
```

STNT1H { <Zt>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 16;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>` Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<Xm>` Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset = X[m];
bits(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
    if LastActiveElement(mask, esize) >= 0 ||
        ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
            CheckSPAlignment();
        base = SP[];
    else
        base = X[n];
src = Z[t];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, Acctype_STREAM] = Elem[src, e, esize];
    offset = offset + 1;
```
STNT1W (vector plus scalar)

Scatter store non-temporal words.

Scatter store non-temporal of words from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

It has encodings from 2 classes: **32-bit unscaled offset** and **64-bit unscaled offset**

### 32-bit unscaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | Pg | Zn | Zt |
```

**msz<1>msz<0>**

STNT1W { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;
integer msize = 32;

### 64-bit unscaled offset

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | Pg | Zn | Zt |
```

**msz<1>msz<0>**

STNT1W { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 32;

**Assembler Symbols**

- `<Zt>` Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zn>` Is the name of the base scalable vector register, encoded in the "Zn" field.
- `<Xm>` Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.
Operation

\begin{verbatim}
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) base = Z[n];
bits(64) offset = X[m];
bits(VL) src = Z[t];
bits(PL) mask = P[g];
bits(64) addr;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize]<msize-1:0>;
\end{verbatim}

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1W (scalar plus immediate)

Contiguous store non-temporal words from vector (immediate index).

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

```
STNT1W { <Zt>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]}
```

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 32;
integer offset = SInt(imm4);

Assembler Symbols

- `<Zt>`: Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Xn|SP>`: Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
- `<imm>`: Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g];

if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
    ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
    CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[;]
  else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];
  src = Z[t];
  addr = base + offset * elements * mbytes;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
    addr = addr + mbytes;
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1W (scalar plus scalar)

Contiguous store non-temporal words from vector (scalar index).

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
[1 1 1 0 0 1 0 1 0 0 0 0 0 0 0 1 1 0 1 1 | Rm | 0 1 1 | Pg | Rn | Zt]

msz<1>msz<0>

STNT1W { <Zt>.S }, <Pg>, [<Xn|SP>, < Xm>, LSL #2]

if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
integer esize = 32;

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bites(64) addr;
bites(64) offset = X[m];
bites(VL) src;
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
if n == 31 then
  if LastActiveElement(mask, esize) >= 0 ||
     ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
      CheckSPAlignment();
based = SP[];
else
  base = X[n];
src = Z[t];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  if ElemP[mask, e, esize] == '1' then
    Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
  offset = offset + 1;
**STR (predicate)**

Store predicate register.

Store a predicate register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated. The store is performed as a stream of bytes containing 8 consecutive predicate bits in ascending element order, without any endian conversion.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm9h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>imm9l</td>
<td>Rn</td>
<td>0</td>
<td>Pt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**STR <Pt>, [<Xn|SP>{{, #<imm>, MUL VL}}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

**Assembler Symbols**

<Pt> Is the name of the scalable predicate transfer register, encoded in the "Pt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

**Operation**

CheckSVEEnabled();
integer elements = PL DIV 8;
bits(PL) src;
bits(64) base;
integer offset = imm * elements;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n];

src = P[t];
boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_NORMAL, TRUE);
for e = 0 to elements-1
    AArch64.MemSingle[base + offset, 1, AccType_NORMAL, aligned] = Elem[src, e, 8];
    offset = offset + 1;
STR (vector)

Store vector register.

Store a vector register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.
The store is performed as a stream of byte elements in ascending element order, without any endian conversion.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | | | | | | | | | | | | | | | | | | | | | | | |
| imm9h | 0 | 1 | 0 | imm9l | |
| Rn | Zt |

STR <Zt>, [<Xn|SP>], #<imm>, MUL VL}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm> Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

CheckSVEEnabled();
integer elements = VL DIV 8;
bits(VL) src;
bits(64) base;
integer offset = imm * elements;

if n == 31 then
  CheckSPAlignment();
  if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
  base = SP[];
else
  if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
  base = X[n];

src = Z[t];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_NORMAL, TRUE);
for e = 0 to elements-1
  AArch64.MemSingle[base + offset, 1, AccType_NORMAL, aligned] = Elem[src, e, 8];
offset = offset + 1;
**SUB (vectors, predicated)**

Subtract vectors (predicated).

Subtract active elements of the second source vector from corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector registers remain unmodified.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Assembler Symbols**

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the “Zdn” field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

**Operation**

```plaintext
CheckSVEEnabled();
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SUB (immediate)**

Subtract immediate (unpredicated).

Subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<imm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|------------------|-----------|-------|--------|------|------|------|-------|
| size 1 0 0 0 1 0 1 | imm8 0 0 0 1 1 1 | sh 0 | Zdn 0 0 0 0 0 1 1 1 1 |

**Assembler Symbols**

- `<Zdn>` Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>` Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
- `<shift>` Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th><code>&lt;shift&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 - imm;
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
\textbf{SUB (vectors, unpredicated)}

Subtract vectors (unpredicated).

Subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

\begin{verbatim}
0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0
\end{verbatim}

\texttt{SUB \langle Zd \rangle, \langle T \rangle, \langle Zn \rangle, \langle T \rangle, \langle Zm \rangle, \langle T \rangle}

\begin{verbatim}
if \text{!HaveSVE()} then UNDEFINED;
integer esize = 8 \ll \text{UInt(size)};
integer n = \text{UInt(Zn)};
integer m = \text{UInt(Zm)};
integer d = \text{UInt(Zd)};
\end{verbatim}

\textbf{Assembler Symbols}

\texttt{\langle Zd \rangle} Is the name of the destination scalable vector register, encoded in the "Zd" field.

\texttt{\langle T \rangle} Is the size specifier, encoded in "size":

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{size} & \texttt{\langle T \rangle} \\
\hline
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\hline
\end{tabular}
\end{center}

\texttt{\langle Zn \rangle} Is the name of the first source scalable vector register, encoded in the "Zn" field.

\texttt{\langle Zm \rangle} Is the name of the second source scalable vector register, encoded in the "Zm" field.

\textbf{Operation}

\begin{verbatim}
CheckSVEEnabled();
integer elements = \text{VL} \div esize;
bits(\text{VL}) operand1 = \text{Z[n]};
bits(\text{VL}) operand2 = \text{Z[m]};
bits(\text{VL}) result;
for e = 0 to elements-1
  bits(esize) element1 = \text{Elem[operand1, e, esize]};
bits(esize) element2 = \text{Elem[operand2, e, esize]};
  \text{Elem[result, e, esize]} = element1 - element2;
\text{Z[d]} = result;
\end{verbatim}

\textbf{Operational information}

If \text{PSTATE.DIT} is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**SUBHNB**

Subtract narrow high part (bottom).

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**SUBHNB** <Zd>, <T>, <Zn>, <Tb>, <Zm>, <Tb>

```plaintext
if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Tb>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer halfesize = esize DIV 2;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 - element2) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros();
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SUBHNT

Subtract narrow high part (top).

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 1  | 0  | 1  | | S  | R  | T  | | Zm | | Zn | | Zd |

SUBHNT $<Zd>.<T>$, $<Zn>.<Tb>$, $<Zm>.<Tb>$

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
n = UInt(Zn);
m = UInt(Zm);
d = UInt(Zd);

Assembler Symbols

$<Zd>$ is the name of the destination scalable vector register, encoded in the “Zd” field.

$<T>$ is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;T&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

$<Zn>$ is the name of the first source scalable vector register, encoded in the “Zn” field.

$<Tb>$ is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;Tb&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$<Zm>$ is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = $Z[n]$;
bits(VL) operand2 = $Z[m]$;
bits(VL) result = $Z[d]$;
n = UInt(elements);

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 - element2) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;
$Z[d]$ = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
SUBR (vectors)

Reversed subtract vectors (predicated).

Reversed subtract active elements of the first source vector from corresponding elements of the second source vector
and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the
destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  if ElemP[mask, e, esize] == '1' then
    Elem[result, e, esize] = element2 - element1;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate
    register contains the same value for each execution.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**SUBR (immediate)**

Reversed subtract from immediate (unpredicated).

Reversed subtract from an unsigned immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00 | 00 | 00 | 00 | 01 | 10 | 11 | 11 | 01 | 00 | 00 | 00 | 00 | 11 | 01 | 00 | 00 | 00 | 10 | 11 | 00 | 01 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    Elem[result, e, esize] = (imm - element1)<esize-1:0>;
Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SUDOT

Signed by unsigned integer indexed dot product.

The signed by unsigned integer indexed dot product instruction computes the dot product of a group of four signed 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four unsigned 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

<imm> Is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UINT(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SUNPKHI, SUNPKLO

Signed unpack and extend half of vector.

Unpack elements from the lowest or highest half of the source vector and then sign-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: **High half** and **Low half**

### High half

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | U | H |

**SUNPKHI** <Zd>, <T>, <Zn>.<Tb>

```c
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
boolean hi = TRUE;
```

### Low half

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 21| 20| 19| 18| 17| 16| 15| 14| 13| 12| 11| 10|  9|  8|  7|  6|  5|  4|  3|  2|  1|  0|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | U | H |

**SUNPKLO** <Zd>.<T>, <Zn>.<Tb>

```c
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
boolean hi = FALSE;
```

### Assembler Symbols

- **<Zd>** is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zn>** is the name of the source scalable vector register, encoded in the "Zn" field.
- **<Tb>** is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>
Operation

**CheckSVEEnabled();**
integer elements = \( VL \) DIV esize;
integer hsize = esize DIV 2;
bits(VL) operand = \( Z[n] \);
bits(VL) result;

for e = 0 to elements-1
   bits(hsize) element = if hi then \( Elem[\text{operand, } e + \text{elements, } \text{hsize}] \) else \( Elem[\text{operand, } e, \text{hsize}] \);
   \( Elem[\text{result, } e, \text{esize}] = \text{Extend}(\text{element, esize, unsigned}); \)
\( Z[d] = \text{result}; \)

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
Signed saturating addition of unsigned value.

Add active unsigned elements of the source vector to the corresponding signed elements of the addend vector, and destructively place the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element’s signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.

SUQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = SignedSat(SInt(element1) + UInt(element2), esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
SXTB, SXTH, SXTW

Signed byte / halfword / word extend (predicated).

Sign-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: Byte, Halfword and Word

**Byte**

```
0 0 0 0 0 1 0 0 | size 0 1 0 0 0 1 0 1 | Pg  Zn  Zd
```

SXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

if ![HaveSVE]() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;

**Halfword**

```
0 0 0 0 0 1 0 0 | size 0 1 0 0 1 0 1 0 | Pg  Zn  Zd
```

SXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

if ![HaveSVE()] then UNDEFINED;
if size != '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;

**Word**

```
0 0 0 0 0 1 0 0 | size 0 1 0 1 0 0 1 0 1 | Pg  Zn  Zd
```

SXTW <Zd>.D, <Pg>/M, <Zn>.D

if ![HaveSVE()] then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

For the halfword variant: is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
  bits(esize) element = Elem[operand, e, esize];
  if Elem[mask, e, esize] == '1' then
    Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Programmable table lookup in single vector table (merging).

Reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements in the first source vector, and places the indexed element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then the corresponding destination vector element is left unchanged.

Since the index values can select any element in a vector this operation is not naturally vector length agnostic.

If !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[d];
for e = 0 to elements-1
    integer element2 = UInt(Elem[operand2, e, esize]);
    if element2 < elements then
        Elem[result, e, esize] = Elem[operand1, element2, esize];
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
TRN1, TRN2 (predicates)

Interleave even or odd elements from two predicates.

Interleave alternating even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: **Even** and **Odd**

**Even**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 | size 1 0 | Pm 0 1 0 1 0 0 0 | Pn 0 | Pd
```

TRN1 <Pd>..<T>, <Pn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;

**Odd**

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 1 | size 1 0 | Pm 0 1 0 1 0 1 0 | Pn 0 | Pd
```

TRN2 <Pd>..<T>, <Pn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;

**Assembler Symbols**

<p>| &lt;Pd&gt; | Is the name of the destination scalable predicate register, encoded in the &quot;Pd&quot; field. |</p>
<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
<th>Is the size specifier, encoded in &quot;size&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>&lt;T&gt;</td>
</tr>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

| <Pn> | Is the name of the first source scalable predicate register, encoded in the "Pn" field. |
| <Pm> | Is the name of the second source scalable predicate register, encoded in the "Pm" field. |
**Operation**

```c
CheckSVEEnabled();
integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

for p = 0 to pairs-1
    `Elem`[result, 2*p+0, esize DIV 8] = `Elem`[operand1, 2*p+part, esize DIV 8];
    `Elem`[result, 2*p+1, esize DIV 8] = `Elem`[operand2, 2*p+part, esize DIV 8];

P[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
TRN1, TRN2 (vectors)

Interleave even or odd elements from two vectors.

Interleave alternating even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

It has encodings from 4 classes: Even, Even (quadwords), Odd and Odd (quadwords)

Even

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
TRN1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Even (quadwords)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |  1 |  Zm | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

```
TRN1 <Zd>.Q, <Zn>.Q, <Zm>.Q
```

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Odd

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |  1 |  Zm | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

```
TRN2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>
```

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Odd (quadwords)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |  1 |  Zm | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UINT(Zn);
integer m = UINT(Zm);
integer d = UINT(Zd);
integer part = 1;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
if VL < esize * 2 then UNDEFINED;
integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Zn;
bits(VL) operand2 = Zm;
bits(VL) result = Zeros();

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**UABA**

Unsigned absolute difference and accumulate.

Compute the absolute difference between unsigned integer values in elements of the second source vector and corresponding elements of the first source vector, and add the difference to the corresponding elements of the destination vector. This instruction is unpredicated.

### Assembler Symbols

- **<Zda>** Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the first source scalable vector register, encoded in the “Zn” field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the “Zm” field.

### Operation

1. **CheckSVEEnabled();**
2. integer esize = 8 << Uint(size);
3. integer n = Uint(Zn);
4. integer m = Uint(Zm);
5. integer da = Uint(Zda);
6. boolean unsigned = TRUE;
7. if !HaveSVE2() then UNDEFINED;
8. integer elements = VL DIV esize;
9. bits(VL) operand1 = Z[n];
10. bits(VL) operand2 = Z[m];
11. bits(VL) result = Z[da];
12. for e = 0 to elements-1
   13.   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   14.   integer element2 = Int(Elem[operand2, e, esize], unsigned);
   15.   bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
   16.   Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
17. Z[da] = result;

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a **MOVPRFX** instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned absolute difference and accumulate long (bottom).

Compute the absolute difference between even-numbered unsigned elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 | size | 0 | Zm | 1 1 0 0 | 1 0 | Zn | Zda
```

UABALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned absolute difference and accumulate long (top).

Compute the absolute difference between odd-numbered unsigned elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | Zm | 1 | 1 | 0 | 0 | 1 | 1 | Zn | Zda |

UABALT <Zda>, <T>, <Zn>, <Tb>, <Zm>, <Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UABD

Unsigned absolute difference (predicated).

Compute the absolute difference between unsigned integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

UABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if Elem[mask, e, esize] == '1' then
        integer absdiff = Abs(element1 - element2);
        Elem[result, e, esize] = absdiff<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09 xml, sve v2020-09 rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UABDLB

Unsigned absolute difference long (bottom).

Compute the absolute difference between the even-numbered unsigned integer values in elements of the second source vector and the corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------------------------|----------------------------------|----------------------------------|----------------------------------|
| 0 1 0 0 1 0 1 0 0 1 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 |
| S U T                           | Zm                               | Zn                               | Zd                               |

UABDLB \(<Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>\)

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
Unsigned absolute difference long (top).

Compute the absolute difference between the odd-numbered unsigned integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

UABDLT $<\text{Zd}>$. $<\text{T}>$, $<\text{Zn}>$. $<\text{Tb}>$, $<\text{Zm}>$. $<\text{Tb}>$

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 $\ll$ UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

$<\text{Zd}>$ Is the name of the destination scalable vector register, encoded in the “Zd” field.

$<\text{T}>$ Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;\text{T}&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

$<\text{Zn}>$ Is the name of the first source scalable vector register, encoded in the “Zn” field.

$<\text{Tb}>$ Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>$&lt;\text{Tb}&gt;$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

$<\text{Zm}>$ Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL $\div$ esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
   integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
   integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
   integer res = Abs(element1 - element2);
   Elem[result, e, esize] = res<esize-1:0>;

Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
• The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UADALP

Unsigned add and accumulate long pairwise.

Add pairs of adjacent unsigned integer values and accumulate the results into the overlapping double-width elements of the destination vector.

```
0 1 0 0 1 0 0 size 0 0 0 1 0 1 1 0 1  P|g  Z|n  Z|da
```

UADALP <Zda>.<T>, <Pg>/M, <Zn>.<Tb>

- \( \text{if} \ !\text{CheckSVEEnabled}() \) then UNDEFINED;
- \( \text{if} \ \text{size} == '00' \) then UNDEFINED;
- integer esize = 8 << \text{UInt}(\text{size});
- integer g = \text{UInt}(Pg);
- integer n = \text{UInt}(Zn);
- integer da = \text{UInt}(Zda);

**Assembler Symbols**

- \(<\text{Zda}>\) Is the name of the second source and destination scalable vector register, encoded in the "Zda" field.
- \(<\text{T}>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<\text{Pg}>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<\text{Zn}>\) Is the name of the first source scalable vector register, encoded in the "Zn" field.
- \(<\text{Tb}>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

**Operation**

\( \text{CheckSVEEnabled}(); \)

integer elements = \text{VL} / \text{DIV} esize;

bits(PL) mask = \text{P}[g];
bits(VL) operand_acc = \text{Z}[da];
bits(VL) operand_src = \text{Z}[n];
bits(VL) result;

for e = 0 to elements-1
    if \text{Elem}[mask, e, esize] == '0' then
        \text{Elem}[result, e, esize] = \text{Elem}[operand_acc, e, esize];
    else
        integer element1 = \text{UInt}(\text{Elem}[operand_src, 2*e + 0, esize DIV 2]);
        integer element2 = \text{UInt}(\text{Elem}[operand_src, 2*e + 1, esize DIV 2]);
        bits(esize) sum = (element1 + element2)<esize-1:0>;
        \text{Elem}[result, e, esize] = \text{Elem}[operand_acc, e, esize] + sum;

\text{Z}[da] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UADDLB**

Unsigned add long (bottom).

Add the corresponding even-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
0 1 0 0 0 1 0 1 | size | 0 0 0 0 1 0 1 0 | Zm | 0 0 0 1 0 1 0 | Zn | Zd
```

**UADDLB** `<Zd>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

- If `!HaveSVE()` then UNDEFINED;
- if size == '00' then UNDEFINED;
- integer esize = 8 << Uint(size);
- integer n = Uint(Zn);
- integer m = Uint(Zm);
- integer d = Uint(Zd);
- integer sel1 = 0;
- integer sel2 = 0;
- boolean unsigned = TRUE;

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Tb>` Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UADDLT

Unsigned add long (top).

Add the corresponding odd-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
0 1 0 0 0 1 0 1 | size | 0  Zm | 0 0 0 0 1 1 | Zn | Zd
```

UADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
```

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

```
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>
```

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
  integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
  integer res = element1 + element2;
  Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UADDV**

Unsigned add reduction to scalar.

Unsigned add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first zero-extended to 64 bits. Inactive elements in the source vector are treated as zero.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | Pg | Zn | Vd |

UADDV <Dd>, <Pg>, <Zn>..<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);

**Assembler Symbols**

<Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer sum = 0;

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = UInt(Elem[operand, e, esize]);
    sum = sum + element;
V[d] = sum<63:0>;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
**UADDWB**

Unsigned add wide (bottom).

Add the even-numbered unsigned elements of the second source vector to the overlapping double-width elements of
the first source vector and place the results in the corresponding double-width elements of the destination vector. This
instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | size | 0 | Zm | 0 | 1 | 0 | 0 | 1 | 0 | Zn | Zd |
| S | U | T |

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

**Operation**

*CheckSVEEnabled*();
integer elements = *VL* DIV esize;
bits(*VL*) operand1 = *Z*[n];
bits(*VL*) operand2 = *Z*[m];
bits(*VL*) result;

for e = 0 to elements-1
    integer element1 = *UInt*(Elem[operand1, e, esize]);
    integer element2 = *UInt*(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

*Z*[d] = result;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
**UADDWT**

Unsigned add wide (top).

Add the odd-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>Zm</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**UADDWT** \(<Zd>\).<T>, \(<Zn>\).<T>, \(<Zm>\).<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;
Z[d] = result;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  o The values of the data supplied in any of its registers.
  o The values of the NZCV flags.
UCVTF

Unsigned integer convert to floating-point (predicated).

Convert to floating-point from the unsigned integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.

It has encodings from 7 classes: 16-bit to half-precision, 32-bit to half-precision, 32-bit to single-precision, 32-bit to double-precision, 64-bit to half-precision, 64-bit to single-precision, and 64-bit to double-precision.

16-bit to half-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 1 0 1 1 0 1 0 0 1 0 1 1 1 0 1 0 0 1 Zd

UCVTF <Zd>.H, <Pg>/M, <Zn>.H
```

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 16;
integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to half-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 1 0 1 1 0 1 0 1 0 1 1 1 1 0 1 1 0 1 Zd

UCVTF <Zd>.H, <Pg>/M, <Zn>.S
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

32-bit to single-precision

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 1 0 1 0 1 1 1 0 0 1 0 1 1 1 1 0 1 1 1 0 Zd

UCVTF
```

Page 2647
if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 32;
integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

### 32-bit to double-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |

### 64-bit to half-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  |

### 64-bit to single-precision

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |
64-bit to double-precision

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 0 1 0 1 1 0 1 0 1</td>
</tr>
<tr>
<td>int_U</td>
</tr>
</tbody>
</table>

UCVTF <Zd>.D, <Pg>/M, <Zn>.D

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
integer s_esize = 64;
integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding);
        Elem[result, e, esize] = ZeroExtend(fpval);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UDIV**

Unsigned divide (predicated).

Unsigned divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |
| R   | U  |
```

**UDIV** `<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size<0>”:
  - size<0> <T>
  - 0 S
  - 1 D
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if ElemP[mask, e, esize] == '1' then
    integer quotient;
    if element2 == 0 then
      quotient = 0;
    else
      quotient = RoundTowardsZero(Real(element1) / Real(element2));
      Elem[result, e, esize] = quotient<esize-1:0>;
    else
      Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDIVR

Unsigned reversed divide (predicated).

Unsigned reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|----------------|
| 0 0 0 0 0 1 0 0 | size | 0 1 0 1 1 1 0 0 0 | Pg | Zm | Zdn       |
| R U                                      |

UDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt; &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 S</td>
</tr>
<tr>
<td>1 D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  integer element2 = Int(Elem[operand2, e, esize], unsigned);
  if ElemP[mask, e, esize] == '1' then
    integer quotient;
    if element1 == 0 then
      quotient = 0;
    else
      quotient = RoundTowardsZero(Real(element2) / Real(element1));
      Elem[result, e, esize] = quotient<esize-1:0>;
      else
      Elem[result, e, esize] = Elem[operand1, e, esize];
        Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDOT (vectors)

Unsigned integer dot product.

The unsigned integer dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. This instruction is unpredicated.

```
UDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>
```

if !HaveSVE() then UNDEFINED;
if size == '0x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
in integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

| <Zda> | Is the name of the third source and destination scalable vector register, encoded in the “Zda” field. |
| <T>   | Is the size specifier, encoded in “size<0>”: |
| size<0> | <T> |
| 0      | S     |
| 1      | D     |

| <Zn> | Is the name of the first source scalable vector register, encoded in the “Zn” field. |
| <Tb> | Is the size specifier, encoded in “size<0>”: |
| size<0> | <Tb> |
| 0      | B     |
| 1      | H     |

| <Zm> | Is the name of the second source scalable vector register, encoded in the “Zm” field. |

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) res = Elem[operand3, e, esize];
  for i = 0 to 3
    integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
    res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UDOT (indexed)

Unsigned integer indexed dot product.

The unsigned integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 0 1 i2 Zm 0 0 0 0 0 1 Zn Zda</td>
</tr>
</tbody>
</table>

size<1>size<0> U


if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0 1 1 i1 Zm 0 0 0 0 0 1 Zn Zda</td>
</tr>
</tbody>
</table>

size<1>size<0> U


if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<i<imm>> For the 32-bit variant: is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

For the 64-bit variant: is the immediate index of a quadtuplet of four 16-bit elements within each 128-bit vector segment, in the range 0 to 1, encoded in the “i1” field.
**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UHADD

Unsigned halving addition.

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|
| 0 1 0 0 0 1 0 0 | size 0 1 0 0 1 1 0 0 | Pg Zm Zdn |
```

UHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(P) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = element1 + element2;
        Elem[result, e, esize] = res<esize:1>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UHSUB

Unsigned halving subtract.

Subtract active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

UHSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = element1 - element2;
        Elem[result, e, esize] = res<esize:1>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UHSUBR

Unsigned halving subtract reversed vectors.

Subtract active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

if CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = element2 - element1;
        Elem[result, e, esize] = res<esize:1>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMAX (vectors)**

Unsigned maximum vectors (predicated).

Determine the unsigned maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

![Vector Register](image)

UMAX \(<Zdn>\), \(<T>\), \(<Pg>/M\), \(<Zdn>\), \(<Zm>\), \(<T>\)

if !`HaveSVE()` then UNDEFINED;
integer esize = 8 << `UInt(size)`;
integer g = `UInt(Pg)`;
integer dn = `UInt(Zdn)`;
integer m = `UInt(Zm)`;
boolean unsigned = TRUE;

**Assembler Symbols**

- `<Zdn>`: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>`: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>`: Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

`CheckSVEEnabled()`;
integer elements = `VL` DIV esize;
bits(`PL`) mask = `P[g]`;
bits(`VL`) operand1 = `Z[dn]`;
bits(`VL`) operand2 = `Z[m]`;
bits(`VL`) result;

for e = 0 to elements-1
    integer element1 = `Int(Elem[operand1, e, esize], unsigned)`;
    integer element2 = `Int(Elem[operand2, e, esize], unsigned)`;
    if `ElemP[mask, e, esize]` == '1' then
        integer maximum = `Max(element1, element2)`;
        `Elem[result, e, esize]` = `maximum<esize-1:0>`;
    else
        `Elem[result, e, esize]` = `Elem[operand1, e, esize]`;

`Z[dn]` = result;

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMAX (immediate)

Unsigned maximum with immediate (unpredicated).

Determine the unsigned maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 | size 1 0 1 0 0 1 1 1 0 | imm8 1 1 1 0 0 1 0 1
```

```
UMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

```c
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

- `<Zdn>`: Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>`: Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<imm>`: Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;
Z[dn] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMAXP

Unsigned maximum pairwise.

Compute the maximum value of each pair of adjacent unsigned integer elements within each source vector, and
interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first
source vector.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|----------------|----------------|----------------|
| 0 1 0 0 0 1 0 0 | size 0 1 0 1 0 1 1 0 1 | Pg | Zm | Zdn |

UMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if !ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = UInt(Elem[operand1, e + 0, esize]);
      element2 = UInt(Elem[operand1, e + 1, esize]);
    else
      element1 = UInt(Elem[operand2, e - 1, esize]);
      element2 = UInt(Elem[operand2, e + 0, esize]);
    integer res = Max(element1, element2);
    Elem[result, e, esize] = res<esize-1:0>;
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMAXV

Unsigned maximum reduction to scalar.

Unsigned maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer n = Uint(Zn);
integer d = Uint(Vd);
boolean unsigned = TRUE;

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the “Vd” field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

<Zn> Is the name of the source scalable vector register, encoded in the “Zn” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer maximum = if unsigned then 0 else -(2^(esize-1));
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    integer element = Int(Elem[operand, e, esize], unsigned);
    maximum = Max(maximum, element);
V[d] = maximum<esize-1:0>;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
**UMIN (vectors)**

Unsigned minimum vectors (predicated).

Determine the unsigned minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | Pg | Zm | Zdn |
```

**Assembler Symbols**

- `<Zdn>` is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer minimum = Min(element1, element2);
        Elem[result, e, esize] = minimum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMIN (immediate)

Unsigned minimum with immediate (unpredicated).

Determine the unsigned minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 0 1 0 0 1 0 1 0 1 0 1 1 1 0 1 1 1 0 0 0 0 1 0 1 1 1 0

imm8   Zdn

UMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMINP**

Unsigned minimum pairwise.

Compute the minimum value of each pair of adjacent unsigned integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 0 0 1 0 0 | size | 0 1 0 1 1 1 1 0 1 | Pg | Zm | Zdn |

**UMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>**

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer m = Uint(Zm);
integer dn = Uint(Zdn);

**Assembler Symbols**

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer element1;
integer element2;
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '0' then
    Elem[result, e, esize] = Elem[operand1, e, esize];
  else
    if IsEven(e) then
      element1 = Uint(Elem[operand1, e + 0, esize]);
      element2 = Uint(Elem[operand1, e + 1, esize]);
    else
      element1 = Uint(Elem[operand2, e - 1, esize]);
      element2 = Uint(Elem[operand2, e + 0, esize]);
    integer res = Min(element1, element2);
    Elem[result, e, esize] = res<esize-1:0>;
  end if
Z[dn] = result;

**Operational information**

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMINV

Unsigned minimum reduction to scalar.

Unsigned minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.

\[
\begin{array}{cccccccccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & Zn & Vd & U
\end{array}
\]

UMINV \(<V><d>, \,<Pg>, \,<Zn>.\,<T>

if ! HaveSVE() then UNDEFINED;
integer esize = 8 \(<Uno\text{t}(size));
integer g = Uint(Pg);
integer n = Uint(Zn);
integer d = Uint(Vd);
boolean unsigned = TRUE;

Assembler Symbols

\(<V>\) Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;V&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<d>\) Is the number \([0-31]\) of the destination SIMD&FP register, encoded in the “Vd” field.

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the “Pg” field.

\(<Zn>\) Is the name of the source scalable vector register, encoded in the “Zn” field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
integer minimum = if unsigned then \((2^\text{esize} - 1)\) else \((2^{(\text{esize}-1)} - 1)\);
for e = 0 to elements-1
    if Elem[\text{mask}, e, esize] == '1' then
        integer element = Int(Elem[\text{operand}, e, esize], unsigned);
        minimum = Min(minimum, element);
V[d] = minimum<esize-1:0>;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.
UMLALB (vectors)

Unsigned multiply-add long to accumulator (bottom).

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  0 1 0 0 0 1 0 0 size | 0 Zm | 0 1 0 0 1 0 | Zn | Zda
```

If !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

```
if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
  bits(esize) product = (element1 * element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] + product;
Z[da] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMLALB (indexed)

Unsigned multiply-add long to accumulator (bottom, indexed).

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|--------------------------|--------------------------|--------------------------|
| 0 1 0 0 0 1 0 0 1 0 1 0 1 | 0 1 0 0 0 1 0 1 | Zm | 0 0 1 0 1 | Zn | Zda |
```


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

### 64-bit

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|--------------------------|--------------------------|--------------------------|--------------------------|
| 0 1 0 0 0 1 0 0 1 0 1 1 | 0 1 1 2 1 | Zm | 0 0 1 0 1 | Zn | Zda |
```

UMLALB `<Zda>.D, <Zn>.S, <Zm>.S[<imm>]`

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
  - For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
- `<imm>` For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.
  - For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.
Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMLALT (vectors)

Unsigned multiply-add long to accumulator (top).

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 size 0 Zm 0 1 0 0 1 1 Zn Zda
```

UMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;
Z[da] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMLALT (indexed)

Unsigned multiply-add long to accumulator (top, indexed).

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 0 1 0 0 0 | 1 0 1 | i3h | Zm | 1 0 1 | i3l | 1 | Zn | Zda |
```

if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

### 64-bit

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 0 1 0 0 0 | 1 1 1 | i2h | Zm | 1 0 1 | i2l | 1 | Zn | Zda |
```

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

### Assembler Symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
- `<imm>` For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.
Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] += element1 * element2;

Z[da] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMLSLB (vectors)

Unsigned multiply-subtract long from accumulator (bottom).

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 | size | 0 | Zm | 0 1 0 1 1 0 | Zn | Zda
```

UMLSLB <Zda>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !CheckSVEEnabled() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

**Assembler Symbols**

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<T>` Is the size specifier, encoded in “size”:
  ```
  size  <T>
  00   RESERVED
  01   H
  10   S
  11   D
  ```
- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Tb>` Is the size specifier, encoded in “size”:
  ```
  size  <Tb>
  00   RESERVED
  01   B
  10   H
  11   S
  ```
- `<Zm>` Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

- `integer elements = VL DIV esize;
  bits(VL) operand1 = Z[n];
  bits(VL) operand2 = Z[m];
  bits(VL) result = Z[da];`
- `for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
  bits(esize) product = (element1 * element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] - product;`
- `Z[da] = result;`

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
  • The MOVPRFX instruction must be unpredicated.
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMLSLB (indexed)

Unsigned multiply-subtract long from accumulator (bottom, indexed).

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 0 0 1 0 1 0 1 i3h Zm 1 0 1 1 i3l 0 Zn Zda</td>
</tr>
</tbody>
</table>


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 0 0 1 0 1 1 i2h Zm 1 0 1 1 i2l 0 Zn Zda</td>
</tr>
</tbody>
</table>

UMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.

<imm> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the “i3h:i3l” fields.

For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the “i2h:i2l” fields.
Operation

\textbf{CheckSVEEnabled}();
integer elements = \texttt{VL} \ DIV (2 * \texttt{esize});
integer eltspersegment = 128 \ DIV (2 * \texttt{esize});
bits(\texttt{VL}) operand1 = \texttt{Z}[n];
bits(\texttt{VL}) operand2 = \texttt{Z}[m];
bits(\texttt{VL}) result = \texttt{Z}[da];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = \texttt{UInt}(\texttt{Elem}[operand1, 2 * e + sel, \texttt{esize}]);
    integer element2 = \texttt{UInt}(\texttt{Elem}[operand2, 2 * s + index, \texttt{esize}]);
    bits(2*\texttt{esize}) product = (element1 * element2)<2*\texttt{esize}-1:0>;
    \texttt{Elem}[result, e, 2*\texttt{esize}] = \texttt{Elem}[result, e, 2*\texttt{esize}] - product;
\texttt{Z}[da] = result;

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
\item The execution time of this instruction is independent of:
    \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
    \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
    \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
    \end{itemize}
\end{itemize}

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction. The \texttt{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \texttt{MOVPRFX} and this instruction is \textbf{UNPREDICTABLE}:
\begin{itemize}
\item The \texttt{MOVPRFX} instruction must be unpredicated.
\item The \texttt{MOVPRFX} instruction must specify the same destination register as this instruction.
\item The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
\end{itemize}
**UMLSLT (vectors)**

Unsigned multiply-subtract long from accumulator (top).

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | Zm | | | | | | | | | | | | | | | | | | | | | | |
| 0  | 1  | 0  | 1  | 1  | 1  | Zn | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| S  | U  | T |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**Assembler Symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[da];
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
  bits(esize) product = (element1 * element2)<esize-1:0>;
  Elem[result, e, esize] = Elem[result, e, esize] - product;
Z[da] = result;

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
  ◦ The MOVPRFX instruction must be unpredicated.
  ◦ The MOVPRFX instruction must specify the same destination register as this instruction.
  ◦ The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMLSLT (indexed)

Unsigned multiply-subtract long from accumulator (top, indexed).

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3h</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3l</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

### 64-bit

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2h</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2l</td>
<td>Zn</td>
<td>Zda</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**UMLSLT** `<Zda>.D, <Zn>.S, <Zm>.S[<imm>]`

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;

### Assembler Symbols

- `<Zda>` is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
  - For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
  - For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the “Zm” field.
- `<imm>` is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  - For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  - For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

`CheckSVEEnabled();`

integer `elements = VL DIV (2 * esize);`

integer `eltspersegment = 128 DIV (2 * esize);`

bits(VL) `operand1 = Z[n];`

bits(VL) `operand2 = Z[m];`

bits(VL) `result = Z[da];`

for `e = 0 to elements-1`

integer `s = e - (e MOD eltspersegment);`

integer `element1 = UInt(Elem[operand1, 2 * e + sel, esize]);`

integer `element2 = UInt(Elem[operand2, 2 * s + index, esize]);`

bits(2*esize) `product = (element1 * element2)<2*esize-1:0>;`

`Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;`

endfor

`Z[da] = result;`

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMMLA**

Unsigned integer matrix multiply-accumulate.

The unsigned integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of unsigned 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

```

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 1 0 1 1 1 0 Zm 1 0 0 1 1 0 Zn Zda
```

**Asmmler Symbols**

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

```

CheckSVEEnabled();
integer segments = VL DIV 128;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(128) op1, op2;
bits(128) res, addend;
for s = 0 to segments-1

    op1    = Elem[operand1, s, 128];
    op2    = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res    = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UMULH (predicated)

Unsigned multiply returning high half (predicated).

Widening multiply unsigned integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|-----------------------------------------------|
| 0  0  0  0  1  0  0 | size 0 1 0 0 1 1 0 0 | Pg 0 1 0 0 0 | Zm 0 0 0 0 0 0 0 0 | Zdn 0 0 0 0 0 0 0 0 |
```

UMULH <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer dn = Uint(Zdn);
integer m = Uint(Zm);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

\[
\begin{array}{|c|c|}
\hline
\text{size} & \text{<T>} \\
\hline
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\hline
\end{array}
\]

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\[
\text{CheckSVEEnabled()};
\]
integer elements = \text{VL} / \text{DIV} esize;
bits(\text{PL}) mask = P[g];
bits(\text{VL}) operand1 = Z[dn];
bits(\text{VL}) operand2 = Z[m];
bits(\text{VL}) result;

for e = 0 to elements-1

\[
\begin{align*}
\text{integer element1} &= \text{Int(Elem[operand1, e, esize], unsigned}); \\
\text{integer element2} &= \text{Int(Elem[operand2, e, esize], unsigned}); \\
\text{if Elem}[mask, e, esize] == '1' \text{ then} \\
&\quad \text{integer product} = (element1 * element2) >> esize; \\
&\quad \text{Elem}[result, e, esize] = \text{product<esize-1:0>}; \\
\text{else} \\
&\quad \text{Elem}[result, e, esize] = \text{Elem}[operand1, e, esize];
\end{align*}
\]

Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

The values of the NZCV flags.

This instruction might be immediately preceded in program order by a **MOVPRFX** instruction. The **MOVPRFX** instruction must conform to all of the following requirements, otherwise the behavior of the **MOVPRFX** and this instruction is **UNPREDICTABLE**:

- The **MOVPRFX** instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The **MOVPRFX** instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UMULH (unpredicated)**

Unsigned multiply returning high half (unpredicated).

Widening multiply unsigned integer values of all elements of the first source vector by corresponding elements of the second source vector and place the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

UMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “size”:
  
<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer product = (element1 * element2) >> esize;
    Elem[result, e, esize] = product<esize-1:0>;
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UMULLB (vectors)

Unsigned multiply long (bottom).

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  |

UMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem(operand1, 2*e + 0, esize DIV 2));
    integer element2 = UInt(Elem(operand2, 2*e + 0, esize DIV 2));
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
UMULLB (indexed)

Unsigned multiply long (bottom, indexed).

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | i3h| Zm | 1  | 1  | 0  | 1  | i3l| Zn | Zd |

size<1>size<0> U T


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;

64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | i2h| Zm | 1  | 1  | 0  | 1  | i2l| Zn | Zd |

size<1>size<0> U T

UMULLB <Zd>.D, <Zn>.S, <Zm>.S[i<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<i3h:i3l> For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \mathop{\text{DIV}} (2 \times \text{esize});
integer eltspersegment = 128 \mathop{\text{DIV}} (2 \times \text{esize});
\text{bits(\texttt{VL}) operand1} = \texttt{Z}[n];
\text{bits(\texttt{VL}) operand2} = \texttt{Z}[m];
\text{bits(\texttt{VL}) result};

\text{for e = 0 to elements-1}
\text{integer s = e - (e MOD eltspersegment);} 
\text{integer element1} = \texttt{UInt}(\texttt{Elem}[\text{operand1}, 2 \times e + \text{sel}, \text{esize}] );
\text{integer element2} = \texttt{UInt}(\texttt{Elem}[\text{operand2}, 2 \times s + \text{index}, \text{esize}] );
\text{integer res} = \text{element1} \times \text{element2};
\texttt{Elem}[result, e, 2\times\text{esize}] = \text{res}<2\times\text{esize}-1:0>;
\text{Z}[^d] = \text{result};

Operational information

If PSTATE.DIT is 1:
\begin{itemize}
\item The execution time of this instruction is independent of:
  \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
  \end{itemize}
\item The response of this instruction to asynchronous exceptions does not vary based on:
  \begin{itemize}
    \item The values of the data supplied in any of its registers.
    \item The values of the NZCV flags.
  \end{itemize}
\end{itemize}
```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

UMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
  integer res = element1 * element2;
  Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
```
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UMULLT (indexed)**

Unsigned multiply long (top, indexed).

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.

The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1 | i3h | Zm | 1 | 1 | 0 | 1 | i3l | 1 | Zn | Zd |

size<1>size<0>  U  T


if !HaveSVE2() then UNDEFINED;
integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;

### 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1 | i2h | Zm | 1 | 1 | 0 | 0 | 0 | i2l | 1 | Zn | Zd |

size<1>size<0>  U  T

**UMULLT** <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

if !HaveSVE2() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
  For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
- `<imm>` For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.
  For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel, esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UQADD (vectors, predicated)

Unsigned saturating addition (predicated).

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \( (2^N) - 1 \). Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UQADD \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>\)

```plaintext
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

- \(<Zdn>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- \(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<Zm>\) Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

```plaintext
CheckSVEEnabled();
it\nteger elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer res = UInt(Sat(element1 + element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

    Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQADD (immediate)

Unsigned saturating add immediate (unpredicated).

Unsigned saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range \(0\) to \((2^N)-1\). This instruction is unpredicated.

The immediate is an unsigned value in the range \(0\) to \(255\), and for element widths of 16 bits or higher it may also be a positive multiple of \(256\) in the range \(256\) to \(65280\).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is “#<imm8>, LSL #8”. However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as “#0, LSL #8”.

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range \(0\) to \(255\), encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh"

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);
Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQADD (vectors, unpredicated)

Unsigned saturating add vectors (unpredicated).

Unsigned saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \((2^N)-1\). This instruction is unpredicated.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  |

UQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);

Z[d] = result;

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECB

Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count.

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 imm4 1 1 1 1 1 1 pattern Rdn | size<1>size<0> sf D U |

UQDECB \(<Wdn>\), \(<pattern>\{, MUL \#<imm>\}\})

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-----------------------------------------------|
| 0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 imm4 1 1 1 1 1 1 pattern Rdn | size<1>size<0> sf D U |

UQDECB \(<Xdn>\), \(<pattern>\{, MUL \#<imm>\}\})

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

\(<Wdn>\)  Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
\(<Xdn>\)  Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
\(<pattern>\)  Is the optional pattern specifier, defaulting to ALL, encoded in "pattern".
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECD (scalar)

Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | imm4 | 1  | 1  | 1  | 1  | 1  | pattern |  Rdn |
| size<1>|size<0> | sf  |  D | U |

UQDECD <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

### 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9  |  8  |  7  |  6  |  5  |  4  |  3  |  2  |  1  |  0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | imm4 | 1  | 1  | 1  | 1  | 1  | pattern |  Rdn |
| size<1>|size<0> | sf  |  D | U |

UQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11110</td>
<td>MUL4</td>
</tr>
<tr>
<td>11111</td>
<td>MUL3</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECD (vector)

Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

Unsaturated decrement vector.

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0 0 1 1 1 0 1 0 1 1</td>
<td>pattern</td>
<td>Zdn</td>
<td></td>
</tr>
</tbody>
</table>

UQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn>     Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01010</td>
<td>VL16</td>
</tr>
<tr>
<td>01011</td>
<td>VL32</td>
</tr>
<tr>
<td>01101</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01110</td>
<td>VL256</td>
</tr>
<tr>
<td>1111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm>     Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQDECH (scalar)**

Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | imm4 | 1  | 1  | 1  | 1  | 1  | 1  | 1  | pattern | Rdn |
| size<1>size<0> | sf | D | U |
```

UQDECH `<Wdn>{, <pattern>{, MUL #<imm>}}`

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

### 64-bit

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | imm4 | 1  | 1  | 1  | 1  | 1  | 1  | 1  | pattern | Rdn |
| size<1>size<0> | sf | D | U |
```

UQDECH `<Xdn>{, <pattern>{, MUL #<imm>}}`

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

**Assembler Symbols**

- `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in "pattern";
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

The immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsaturated decrementing vector by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit unsaturated integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
  integer element1 = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQDECP (scalar)

Unsigned saturating decrement scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 0   | 1   | 1   | 1   | 1   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   |

UQDECP <Wdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 63  | 62  | 61  | 60  | 59  | 58  | 57  | 56  | 55  | 54  | 53  | 52  | 51  | 50  | 49  | 48  | 47  | 46  | 45  | 44  | 43  | 42  | 41  | 40  | 39  | 38  | 37  | 36  | 35  | 34  | 33  | 32  | 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 21  | 20  | 19  | 18  | 17  | 16  | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 0   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 0   | 1   | 1   | 1   | 1   | 0   | 0   | 0   | 1   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 0   |

UQDECP <Xdn>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.
<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element - count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECP (vector)

Unsigned saturating decrement vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Pm |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Zdn |

UQDECP <Zdn>.<T>, <Pm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
  if ElemP[operand2, e, esize] == '1' then
    count = count + 1;
for e = 0 to elements-1
  integer element = Int(Elem[operand1, e, esize], unsigned);
  (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQDECW (scalar)

Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
</tr>
</tbody>
</table>

UQDECW <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>size&lt;1&gt;size&lt;0&gt;</td>
</tr>
</tbody>
</table>

UQDECW <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

UQDECW (scalar)
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1110</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQDECW (vector)

Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 1 0 0 1 0 imm4 1 1 0 0 1 1 pattern Zdn

UQDECW <Zdn>.S{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

```
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#imm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#imm5</td>
</tr>
<tr>
<td>10110</td>
<td>#imm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#imm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#imm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#imm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>
```

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.
Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
   integer element1 = Int(Elem[operand1, e, esize], unsigned);
   (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count.

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | imm4 | 1 | 1 | 1 | 1 | 0 | 1 | pattern | Rdn |

size<1> size<0> sf D U

UQINCB <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | imm4 | 1 | 1 | 1 | 1 | 0 | 1 | pattern | Rdn |

size<1> size<0> sf D U

UQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11001</td>
<td>MUL4</td>
</tr>
<tr>
<td>11010</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCD (scalar)

Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register’s unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

### 32-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 1 0 0 1 1 1 0</th>
<th>imm4</th>
<th>1 1 1 1 0 1</th>
<th>pattern</th>
<th>Rdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
<td>sf</td>
<td>D</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

UQINCD <Wdn>{, <pattern>{, MUL #<imm>}}

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

### 64-bit

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>0 0 0 0 1 0 0 1 1 1</th>
<th>imm4</th>
<th>1 1 1 1 0 1</th>
<th>pattern</th>
<th>Rdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>size&lt;1&gt;</td>
<td>size&lt;0&gt;</td>
<td>sf</td>
<td>D</td>
<td>U</td>
<td></td>
</tr>
</tbody>
</table>

UQINCD <Xdn>{, <pattern>{, MUL #<imm>}}

```plaintext
if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

**Assembler Symbols**

- <Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- <pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”: 
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQINCD (vector)**

Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count.

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size&lt;1&gt;</th>
<th>size&lt;0&gt;</th>
<th>D</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>1</td>
<td>1</td>
<td>1 0</td>
<td>imm4</td>
</tr>
</tbody>
</table>

UQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

**Assembler Symbols**

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
**Operation**

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQINCH (scalar)

Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | imm4 | 1 | 1 | 1 | 1 | 0 | 0 | pattern | Rdn |
| size<1>size<0> | sf | D | U |

UQINCH <Wdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | imm4 | 1 | 1 | 1 | 1 | 0 | 1 | pattern | Rdn |
| size<1>size<0> | sf | D | U |

UQINCH <Xdn>{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in "pattern":

UQINCH (scalar)
<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11001</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

**Operation**

```c
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;
integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
```

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQINCH (vector)

Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count.

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|-------------------------------|--------------------------|
| 0 0 0 0 0 1 0 0 0 1 1 0 1imm4 1 1 0 0 0 1patternZdn | D U                           |

UQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.
<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01010</td>
<td>VL16</td>
</tr>
<tr>
<td>01011</td>
<td>VL32</td>
</tr>
<tr>
<td>01101</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01110</td>
<td>VL256</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**UQINCP (scalar)**

Unsigned saturating increment scalar by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

It has encodings from 2 classes: **32-bit** and **64-bit**

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**UQINCP** `<Wdn>`, `<Pm>`.<**T>**

if !**HaveSVE**() then UNDEFINED;

integer esize = 8 << **UInt**(size);
integer m = **UInt**(Pm);
integer dn = **UInt**(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;

### 64-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**UQINCP** `<Xdn>`, `<Pm>`.<**T>**

if !**HaveSVE**() then UNDEFINED;

integer esize = 8 << **UInt**(size);
integer m = **UInt**(Pm);
integer dn = **UInt**(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;

### Assembler Symbols

**<Wdn>** is the 32-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

**<Xdn>** is the 64-bit name of the source and destination general-purpose register, encoded in the “Rdn” field.

**<Pm>** is the name of the source scalable predicate register, encoded in the "Pm" field.

**<T>** is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn];
bits(PL) operand2 = P[m];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);
UQINCP (vector)

Unsigned saturating increment vector by count of true predicate elements.

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element unsigned integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) operand2 = P[m];
bits(VL) result;
integer count = 0;
for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;
for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], '-') = SatQ(element + count, esize, unsigned);

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQINCW (scalar)

Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

```assembly
UQINCW <Wdn>{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;

64-bit

```assembly
UQINCW <Xdn>{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;

Assembler Symbols

- `<Wdn>` Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<Xdn>` Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
- `<pattern>` Is the optional pattern specifier, defaulting to ALL, encoded in "pattern";
Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn];
bits(ssize) result;
integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);  
X[dn] = Extend(result, 64, unsigned);
```
UQINCW (vector)

Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count.

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:
* A fixed number (VL1 to VL256)
* The largest power of two (POW2)
* The largest multiple of three or four (MUL3 or MUL4)
* All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 0 | imm4 | 1 1 0 0 | pattern | Zdn
```

UQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the “Zdn” field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

<table>
<thead>
<tr>
<th>pattern</th>
<th>&lt;pattern&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>POW2</td>
</tr>
<tr>
<td>00001</td>
<td>VL1</td>
</tr>
<tr>
<td>00010</td>
<td>VL2</td>
</tr>
<tr>
<td>00011</td>
<td>VL3</td>
</tr>
<tr>
<td>00100</td>
<td>VL4</td>
</tr>
<tr>
<td>00101</td>
<td>VL5</td>
</tr>
<tr>
<td>00110</td>
<td>VL6</td>
</tr>
<tr>
<td>00111</td>
<td>VL7</td>
</tr>
<tr>
<td>01000</td>
<td>VL8</td>
</tr>
<tr>
<td>01001</td>
<td>VL16</td>
</tr>
<tr>
<td>01010</td>
<td>VL32</td>
</tr>
<tr>
<td>01011</td>
<td>VL64</td>
</tr>
<tr>
<td>01100</td>
<td>VL128</td>
</tr>
<tr>
<td>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>0111x</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x0x1</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11101</td>
<td>MUL4</td>
</tr>
<tr>
<td>11110</td>
<td>MUL3</td>
</tr>
<tr>
<td>11111</td>
<td>ALL</td>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the “imm4” field.
Operation

CheckSVEEnabled();
integer elements = \text{VL} \text{ DIV } \text{esize};
integer count = \text{DecodePredCount}(\text{pat}, \text{esize});
bits(\text{VL}) \text{ operand1} = Z[dn];
bits(\text{VL}) \text{ result};

for \( e = 0 \) to \( \text{elements} - 1 \)
    integer \text{element1} = \text{Int(Elem}[\text{operand1}, e, \text{esize}], \text{unsigned});
    \text{(Elem}[\text{result}, e, \text{esize}], -) = \text{SatQ}(\text{element1} + (\text{count} * \text{imm}), \text{esize}, \text{unsigned});
\text{Z[dn]} = \text{result};

Operational information

This instruction might be immediately preceded in program order by a \text{MOVPRFX} instruction. The \text{MOVPRFX} instruction must conform to all of the following requirements, otherwise the behavior of the \text{MOVPRFX} and this instruction is \text{UNPREDICTABLE}:

- The \text{MOVPRFX} instruction must be unpredicated.
- The \text{MOVPRFX} instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQRSHL

Unsigned saturating rounding shift left by vector (predicated).

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \(2^N-1\). Inactive elements in the destination vector register remain unmodified.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>size</th>
<th>0 0 1 0 1 0 0</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>R</td>
<td>N</td>
<td>U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UQRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element = UInt(Elem[operand1, e, esize]);
  integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
  integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
  if Elem[mask, e, esize] == '1' then
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQRSHLR

Unsigned saturating rounding shift left reversed vectors (predicated).

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to \(2^N\)-1. Inactive elements in the destination vector register remain unmodified.

An example in assembly language:

```assembly
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[m];
bits(VL) operand2 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer element = UInt(Elem[operand1, e, esize]);
  integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
  integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
  if Elem[mask, e, esize] == '1' then
    integer res = (element + round_const) << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
• The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQRSHRNB

Unsigned saturating rounding shift right narrow by immediate (bottom).

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
0 1 0 0 0 1 0 1 0 tzh 1 tzs 1 imm3 0 0 1 1 1 0 Zn Zd
```

UQRSHRNB <Zd>,<T>, <Zn>,<Tb>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tzh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the “Zd” field.
<T>   Is the size specifier, encoded in “tzh:tszl”:

<table>
<thead>
<tr>
<th>tzh</th>
<th>tzs</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn>  Is the name of the source scalable vector register, encoded in the “Zn” field.
<Tb>  Is the size specifier, encoded in “tzh:tszl”:

<table>
<thead>
<tr>
<th>tzh</th>
<th>tzs</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = (UInt(element) + round_const) >> shift;
  Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros();

Z[d] = result;
UQRSHRNT

Unsigned saturating rounding shift right narrow by immediate (top).

Shift each unsigned integer value in the source vector elements by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

### Assembler Symbols

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Tb>** Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<const>** Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

### Operation

- `CheckSVEEnabled();`
- `integer elements = VL DIV (2 * esize);`
- `bits(VL) operand = Z[n];`
- `bits(VL) result = Z[d];`
- `integer round_const = 1 << (shift-1);`

  for e = 0 to elements-1
  
  \[
  \text{bits}(2*\text{esize}) \text{ element} = \text{Elem}[\text{operand}, e, 2*\text{esize}];
  \]
  \[
  \text{integer res} = (\text{UInt}(\text{element}) + \text{round_const}) >> \text{shift};
  \]
  \[
  \text{Elem}[\text{result}, 2^e + 1, \text{esize}] = \text{UnsignedSat}(\text{res}, \text{esize});
  \]
  
  \[\text{Z}[d] = \text{result};\]
UQSHL (immediate)

Unsigned saturating shift left by immediate.

Shift left by immediate each active unsigned element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
|   |   | 0 0 0 0 0 0 1 0 0 | tszh | 0 0 | 0 1 1 1 1 0 0 | Pg | tszl | imm3 | Zdn |

UQSHL <Zdn>,<T>, <Pg>/M, <Zdn>,<T>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  if Elem[mask, e, esize] == '1' then
    integer res = element1 << shift;
    Elem[result, e, esize] = UnsignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned saturating shift left by vector (predicated).

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \((2^N)-1\). Inactive elements in the destination vector register remain unmodified.

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

UQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```plaintext
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:
  - size `<T>`
    - 00: B
    - 01: H
    - 10: S
    - 11: D
- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    if Elem(mask, e, esize) == '1'
        integer res = element << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned saturating shift left reversed vectors (predicated).

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element’s unsigned integer range \(0 \text{ to } (2^N)-1\). Inactive elements in the destination vector register remain unmodified.

UQSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[m];
bits(VL) operand2 = Z[dn];
bits(VL) result;

for e = 0 to elements - 1
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    if Elem[mask, e, esize] == '1' then
        integer res = element << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned saturating shift right narrow by immediate (bottom).

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while设置 the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

```
if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = UInt(element) >> shift;
  Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
  Elem[result, 2*e + 1, esize] = Zeros();
Z[d] = result;
```

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Unsigned saturating shift right narrow by immediate (top).

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to \((2^N)-1\). The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

\[
\begin{array}{cccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & tszh & 1 & tszl & imm3 & 0 & 0 & 1 & 1 & 0 & 1 & Zn & Zd
\end{array}
\]

UQSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n =(UInt)(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Tb> Is the size specifier, encoded in "tszh:tszl":

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

CheckSVEEnabled();
integer elements = \(\text{VL} \div (2 \times \text{esize})\);
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements - 1
  bits(2*esize) element = Elem[operand, e, 2*esize];
  integer res = UInt(element) >> shift;
  Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);
Z[d] = result;
**UQSUB (vectors, predicated)**

Unsigned saturating subtraction (predicated).

Subtract active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \((2^N)-1\). Inactive elements in the destination vector register remain unmodified.

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | size | 0 | 1 | 1 | 0 | 1 | 1 | 0 | P | g | Zm | Zdn |

UQSUB <Zdn>..<T>, <Pg>/M, <Zdn>..<T>, <Zm>..<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

**Operation**

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer res = UInt(Sat(element1 - element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQSUB (immediate)

Unsigned saturating subtract immediate (unpredicated).

Unsigned saturating subtract an unsigned immediate from each element of the source vector, and destructively place
the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's
unsigned integer range 0 to \(2^N-1\). This instruction is unpredicated.
The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a
positive multiple of 256 in the range 256 to 65280.
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is
specified is ">#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit
value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 size 1 0 0 1 1 1 1 1 sh imm8 Zdn
```

UQSUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

if !HaveSVE() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = TRUE;

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.
<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in "sh":

<table>
<thead>
<tr>
<th>sh</th>
<th>&lt;shift&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LSL #0</td>
</tr>
<tr>
<td>1</td>
<td>LSL #8</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);
Z[dn] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction
must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is
UNPREDICTABLE:
• The `MOVPRFX` instruction must be unpredicated.
• The `MOVPRFX` instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
UQSUB (vectors, unpredicated)

Unsigned saturating subtract vectors (unpredicated).

Unsigned saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to \((2^N)-1\). This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zn</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UQSUB \(<Zd>/.T>, \ <Zn>/.T>, \ <Zm>/.T>

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 << \text{UInt}(size);
integer n = \text{UInt}(Zn);
integer m = \text{UInt}(Zm);
integer d = \text{UInt}(Zd);
boolean unsigned = TRUE;

Assembler Symbols

\(<Zd>\quad \text{Is the name of the destination scalable vector register, encoded in the "Zd" field.}\)
\(<T>\quad \text{Is the size specifier, encoded in "size":}\)

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Zn>\quad \text{Is the name of the first source scalable vector register, encoded in the "Zn" field.}\)
\(<Zm>\quad \text{Is the name of the second source scalable vector register, encoded in the "Zm" field.}\)

Operation

\(\text{CheckSVEEnabled}();\)
integer elements = \text{VL} \div \text{esize};
bits(\text{VL}) operand1 = Z[n];
bits(\text{VL}) operand2 = Z[m];
bits(\text{VL}) result;
for \(e = 0\) to elements-1
  integer element1 = \text{Int}(\text{Elem}[\text{operand1}, e, \text{esize}], \text{unsigned});
  integer element2 = \text{Int}(\text{Elem}[\text{operand2}, e, \text{esize}], \text{unsigned});
  (\text{Elem}[\text{result}, e, \text{esize}], -) = \text{SatQ}(\text{element1} - \text{element2}, \text{esize}, \text{unsigned});
Z[d] = \text{result};
UQSUBR

Unsigned saturating subtraction reversed vectors (predicated).

Subtract active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \((2^N)-1\). Inactive elements in the destination vector register remain unmodified.

```
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

|  0 1 0 0 |  0 1 1 1 1 1 0 0 |

Pg  Zm  Zdn

S  U
```

UQSUBR \(<Zdn>.<T>, \langle Pg\>/M, \langle Zdn\>.<T>, \langle Zm\>.<T>\)

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;

**Assembler Symbols**

\(<Zdn>\) Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

\(<T>\) Is the size specifier, encoded in "size":

| size \(<T>\) |
|---|---|---|---|---|---|---|---|
| 00 | B |
| 01 | H |
| 10 | S |
| 11 | D |

\(<Pg>\) Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\(<Zm>\) Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  integer element2 = UInt(Elem[operand2, e, esize]);
  if ElemP[mask, e, esize] == '1' then
    integer res = UInt(Sat(element2 - element1, esize, unsigned));
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elema[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
Unsigned saturating extract narrow (bottom).

Saturate the unsigned integer value in each source element to half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.

```
0 1 0 0 1 0 0 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 Zn 1 0

Zd T U
```

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '001' esize = 16;
  when '010' esize = 32;
  when '100' esize = 64;
otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd>    Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T>      Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>x</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

<Zn>    Is the name of the source scalable vector register, encoded in the "Zn" field.
<Tb>     Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>x</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) result;
integer halfesize = esize DIV 2;
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  bits(halfesize) res = UnsignedSat(element1, halfesize);
  Elem[result, 2*e + 0, halfesize] = res;
  Elem[result, 2*e + 1, halfesize] = Zeros();
Z[d] = result;
Unsigned saturating extract narrow (top).

Saturate the unsigned integer value in each source element to half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.

\[
\begin{array}{cccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & Zn & Zd & \\
\end{array}
\]

\[<Zd>..<T>, <Zn>..<Tb>\]

if !\texttt{HaveSVE2}() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '001' esize = 16;
  when '010' esize = 32;
  when '100' esize = 64;
  otherwise UNDEFINED;
integer n = \texttt{UInt}(Zn);
integer d = \texttt{UInt}(Zd);

\section*{Assembler Symbols}

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “tszh:tszl”:

  \[
  \begin{array}{ccc}
  tszh & tszl & <T> \\
  0 & 00 & \text{RESERVED} \\
  0 & 01 & B \\
  0 & 10 & H \\
  x & 11 & \text{RESERVED} \\
  1 & 00 & S \\
  1 & 01 & \text{RESERVED} \\
  1 & 10 & \text{RESERVED} \\
  \end{array}
  \]

- **<Zn>** Is the name of the first source scalable vector register, encoded in the "Zn" field.
- **<Tb>** Is the size specifier, encoded in “tszh:tszl”:

  \[
  \begin{array}{ccc}
  tszh & tszl & <Tb> \\
  0 & 00 & \text{RESERVED} \\
  0 & 01 & H \\
  0 & 10 & S \\
  x & 11 & \text{RESERVED} \\
  1 & 00 & D \\
  1 & 01 & \text{RESERVED} \\
  1 & 10 & \text{RESERVED} \\
  \end{array}
  \]

\section*{Operation}

\texttt{CheckSVEEnabled}();
integer elements = \texttt{VL} DIV esize;
bits(\texttt{VL}) operand1 = Z[n];
bits(\texttt{VL}) result = Z[d];
integer halfesize = esize DIV 2;
for e = 0 to elements-1
  integer element1 = \texttt{UInt}(Elem[operand1, e, esize]);
bits(halfesize) res = \texttt{UnsignedSat}(element1, halfesize);
Elem[result, 2*e + 1, halfesize] = res;
Z[d] = result;
URECPE

Unsigned reciprocal estimate (predicated).

Find the approximate reciprocal of each active unsigned element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector remain unmodified.

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Pg  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Zn  | 0  | 0  | 0  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Zd  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

URECPE <Zd>.S, <Pg>/M, <Zn>.S

if !HaveSVE2() then UNDEFINED;
if size != '10' then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = UnsignedRecipEstimate(element);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**URHADD**

Unsigned rounding halving addition.

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | Pg | Zm | Zdn |

**URHADD** <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```plaintext
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

**Assembler Symbols**

- **<Zdn>** Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- **<T>** Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zm>** Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
integer round_const = 1;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if Elem[mask, e, esize] == '1' then
        integer res = element1 + element2 + round_const;
        Elem[result, e, esize] = res<esize:1>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**URSHL**

Unsigned rounding shift left by vector (predicated).

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 1 0 0 0 1 0 0 | size | 0 0 | 0 0 | 1 | 1 | 1 | 0 0 | Pg | Zm | Zdn |
```

URSHL <Zdn><T>, <Pg>/M, <Zdn><T>, <Zm><T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);

**Assembler Symbols**

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    if Elem[mask, e, esize] == '1' then
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URSHLR**

Unsigned rounding shift left reversed vectors (predicated).

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.

-----

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[m];
bits(VL) operand2 = Z[dn];
bits(VL) result;
for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]);
    integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
    integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
    if Elem[mask, e, esize] == '1' then
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
URSHR

Unsigned rounding shift right by immediate.

Shift right by immediate each active unsigned element of the source vector, and destructively place the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|---|---|---|---|
| 0 0 0 0 0 0 1 0 0 | tsh | 0 0 | 1 1 | 0 1 1 0 0 | Pg | tszl | imm3 | Zdn |

URSHR <Zdn>.<<T>>, <Pg>/M, <Zdn>.<<T>>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(4) tsize = tsh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tsh:tszl”:

<table>
<thead>
<tr>
<th>tsh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsh:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(PL) mask = P[g];
bits(VL) result;
integer round_const = 1 << (shift-1);
for e = 0 to elements-1
  integer element1 = UInt(Elem[operand1, e, esize]);
  if ElemP[mask, e, esize] == '1' then
    integer res = (element1 + round_const) >> shift;
    Elem[result, e, esize] = res<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**URSQRTE**

Unsigned reciprocal square root estimate (predicated).

Find the approximate reciprocal square root of each active unsigned element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | Pg | Zn | Zd |

**URSQRTE** <Zd>.S, <Pg>/M, <Zn>.S

if !HaveSVE2() then UNDEFINED;
if size != '10' then UNDEFINED;
integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);

**Assembler Symbols**

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<Pg>** Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.

**Operation**

- **CheckSVEEnabled()**;
- integer elements = VL DIV esize;
- bits(PL) mask = P[g];
- bits(VL) operand = Z[n];
- bits(VL) result = Z[d];
- for e = 0 to elements-1
  - bits(esize) element = Elem[operand, e, esize];
  - if Elem[mask, e, esize] == '1' then
    - Elem[result, e, esize] = UnsignedRSqrtEstimate(element);
- Z[d] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**URSRA**

Unsigned rounding shift right and accumulate (immediate).

Shift right by immediate each unsigned element of the source vector, inserting zeroes, and add the rounded intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| 0 | 1 | 0 | 0 | 1 | 0 | 1 | tszh | 0 | tszl | imm3 | 1 | 1 | 1 | 0 | 1 | 1 | Zn | Zda |
| R | U |

**URSRA <Zda>,<T>, <Zn>,<T>, #<const>

if !HaveSVEz() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);

**Assembler Symbols**

- **<Zda>** Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- **<T>** Is the size specifier, encoded in “tszh:tszl”:
<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>
- **<Zn>** Is the name of the first source scalable vector register, encoded in the “Zn” field.
- **<const>** Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

**Operation**

`CheckSVEEnabled();`
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[da];
bits(VL) result;
integer round_const = 1 << (shift - 1);
for e = 0 to elements-1
  integer element = (UInt(Elem[operand1, e, esize]) + round_const) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
Z[da] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
USDOT (vectors)

Unsigned by signed integer dot product.

The unsigned by signed integer dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  |


if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.
<Zm> Is the name of the second source scalable vector register, encoded in the “Zm” field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
USDOT (indexed)

Unsigned by signed integer indexed dot product.

The unsigned by signed integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 1 0 1 1 0 Zm 0 0 0 1 1 0 Zn 0 Zda
size<1>size<0> U
```


```markdown
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

- `<Zda>` Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
- `<Zn>` Is the name of the first source scalable vector register, encoded in the “Zn” field.
- `<Zm>` Is the name of the second source scalable vector register Z0-Z7, encoded in the “Zm” field.
- `<imm>` Is the immediate index of a quadtuplet of four 8-bit elements within each 128-bit vector segment, in the range 0 to 3, encoded in the “i2” field.

Operation

```markdown
CheckSVEEnabled();
integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result;
for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;
Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
USHLLB

Unsigned shift left long by immediate (bottom).

Shift left by immediate each even-numbered unsigned element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1 1</td>
</tr>
</tbody>
</table>

USHLLB <Zd>,<T>, <Zn>,<Tb>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td></td>
<td>RESERVED</td>
</tr>
<tr>
<td>0 01</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>0 1x</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>1 xx</td>
<td></td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td></td>
<td>RESERVED</td>
</tr>
<tr>
<td>0 01</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>0 1x</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>1 xx</td>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

CheckSVE2Enabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element = Elem[operand, 2*e + 0, esize];
  integer shifted_value = UInt(element) << shift;
  Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;
Z[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**USHLLT**

Unsigned shift left long by immediate (top).

Shift left by immediate each odd-numbered unsigned element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 0 1 0 1 0 | tszh | tszl | imm3 | 1 0 1 0 | 1 1 | Zn | Zd |

**USHLLT <Zd>..<T>, <Zn>.<Tb>, #<const>**

if !HaveSVE2() then UNDEFINED;
bits(3) tsize = tszh:tszl;
case tsize of
  when '000' UNDEFINED;
  when '001' esize = 8;
  when '01x' esize = 16;
  when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>1 xx</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>1 xx</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

**Operation**

CheckSVEEnabled();
integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element = Elem[operand, 2*e + 1, esize];
  integer shifted_value = UInt(element) << shift;
  Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;
Z[d] = result;
Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
USMMLA

Unsigned by signed integer matrix multiply-accumulate.

The unsigned by signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

This instruction is unpredicated. ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 1 | 1 | 0 0 | Zm | 1 0 0 1 1 0 | Zn | Zda

if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

\[
\text{CheckSVEEnabled}(); \\
\text{integer segments} = \text{VL} \div 128; \\
\text{bits(\text{VL}) operand1} = Z[n]; \\
\text{bits(\text{VL}) operand2} = Z[m]; \\
\text{bits(\text{VL}) operand3} = Z[da]; \\
\text{bits(\text{VL}) result} = \text{Zeros}(); \\
\text{bits(128) op1, op2}; \\
\text{bits(128) res, addend}; \\
\text{for s = 0 to segments-1} \\
\quad \text{op1} = \text{Elem}[\text{operand1}, s, 128]; \\
\quad \text{op2} = \text{Elem}[\text{operand2}, s, 128]; \\
\quad \text{addend} = \text{Elem}[\text{operand3}, s, 128]; \\
\quad \text{res} = \text{MatMulAdd}(\text{addend}, \text{op1}, \text{op2}, \text{op1_unsigned}, \text{op2_unsigned}); \\
\quad \text{Elem}[\text{result}, s, 128] = \text{res}; \\
\]

\[Z[\text{da}] = \text{result}\

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
**USQADD**

Unsigned saturating addition of signed value.

Add active signed elements of the source vector to the corresponding unsigned elements of the addend vector, and destructively place the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element’s unsigned integer range 0 to \(2^N-1\). Inactive elements in the destination vector register remain unmodified.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|----------------|----------------|----------------|
| 0 1 0 0 0 1 0 0 | size | 0 1 1 1 0 1 1 0 |

USQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << Uint(size);
integer g = Uint(Pg);
integer dn = Uint(Zdn);
integer m = Uint(Zm);
```

**Assembler Symbols**

- `<Zdn>` Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- `<T>` Is the size specifier, encoded in “size”:

```
size  | <T>
-----|-----
00   | B
01   | H
10   | S
11   | D

<Pg>` Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = UnsignedSat(UInt(element1) + SInt(element2), esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];
Z[dn] = result;
```

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
USRA

Unsigned shift right and accumulate (immediate).

Shift right by immediate each unsigned element of the source vector, inserting zeroes, and add the truncated intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------------------------|--------------------------------------------------|
| 0 1 0 0 0 1 0 1 tszh 0 tszl imm3 1 1 1 0 0 1 | Zn Zda                                           |

USRA <Zda><T>, <Zn><T>, #<const>

if !HaveSVE2() then UNDEFINED;

bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the “Zn” field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[da];
bits(VL) result;
for e = 0 to elements-1
  integer element = UInt(Elem[operand1, e, esize]) >> shift;
  Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;
Z[da] = result;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
USUBLB

Unsigned subtract long (bottom).

Subtract the even-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

```
0 1 0 0 0 1 1 0 | size | Zm | 0 0 0 1 1 0 | Zn | Zd
S U T
```

**USUBLB** <Zd>..<T>, <Zn>..<Tb>, <Zm>..<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = TRUE;

**Assembler Symbols**

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
```
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USUBLT

Unsigned subtract long (top).

Subtract the odd-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

Assembler Symbols

<Zd> is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

<Zm> is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;
Z[d] = result;
**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
USUBWB

Unsigned subtract wide (bottom).

Subtract the even-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 1 0 0 0 1 0 1 | size | 0 | Zm | 0 1 0 1 1 0 | Zn | Zd |
```

USUBWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

- if !HaveSVE2() then UNDEFINED;
- if size == '00' then UNDEFINED;
- integer esize = 8 << UInt(size);
- integer n = UInt(Zn);
- integer m = UInt(Zm);
- integer d = UInt(Zd);

Assembler Symbols

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Tb>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

Operation

- CheckSVEEnabled();
- integer elements = VL DIV esize;
- bits(VL) operand1 = Z[n];
- bits(VL) operand2 = Z[m];
- bits(VL) result;
- for e = 0 to elements-1
  - integer element1 = UInt(Elem[operand1, e, esize]);
  - integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
  - Elem[result, e, esize] = (element1 - element2)<esize-1:0>;
- Z[d] = result;

Operational information

- If PSTATE.DIT is 1:
  - The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.
**USUBWT**

Unsigned subtract wide (top).

Subtract the odd-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated. This instruction is unpredicated.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Zm</td>
<td>Zn</td>
<td>Zd</td>
<td>S</td>
<td>U</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**USUBWT** `<Zd>`, `<T>`, `<Zn>`, `<T>`, `<Zm>`, `<Tb>`

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);

**Assembler Symbols**

- `<Zd>` Is the name of the destination scalable vector register, encoded in the "Zd" field.
- `<T>` Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<Zn>` Is the name of the first source scalable vector register, encoded in the "Zn" field.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Tb>` Is the size specifier, encoded in "size":
  
<table>
<thead>
<tr>
<th>size</th>
<th><code>&lt;Tb&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>

**Operation**

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;
Z[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UUNPKHI, UUNPKLO**

Unsigned unpack and extend half of vector.

Unpack elements from the lowest or highest half of the source vector and then zero-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.

It has encodings from 2 classes: **High half** and **Low half**

### High half

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|----------------|----------------|
| 0 0 0 0 0 0 1 0 1 | size | 1 1 0 0 1 | Zn | Zd |
```

UUNPKHI <Zd>.<T>, <Zn>.<Tb>

```java
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = TRUE;
```

### Low half

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------------------------------------|----------------|----------------|
| 0 0 0 0 0 0 1 0 1 | size | 1 1 0 0 1 | Zn | Zd |
```

UUNPKLO <Zd>.<T>, <Zn>.<Tb>

```java
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = FALSE;
```

### Assembler Symbols

- **<Zd>** Is the name of the destination scalable vector register, encoded in the "Zd" field.
- **<T>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.
- **<Tb>** Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>H</td>
</tr>
<tr>
<td>11</td>
<td>S</td>
</tr>
</tbody>
</table>
Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
integer hsize = esize DIV 2;
bits(VL) operand = Z[n];
bits(VL) result;

for e = 0 to elements-1
  bits(hsize) element = if hi then Elem[operand, e + elements, hsize] else Elem[operand, e, hsize];
  Elem[result, e, esize] = Extend(element, esize, unsigned);

Z[d] = result;

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**UXTB, UXTH, UXTW**

Unsigned byte / halfword / word extend (predicated).

Zero-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

It has encodings from 3 classes: **Byte**, **Halfword** and **Word**

**Byte**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UXTB**<sub>Zd</sub>.<sub>T</sub>, <sub>Pg</sub>/M, <sub>Zn</sub>.<sub>T</sub>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;

**Halfword**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UXTH**<sub>Zd</sub>.<sub>T</sub>, <sub>Pg</sub>/M, <sub>Zn</sub>.<sub>T</sub>

if !HaveSVE() then UNDEFINED;
if size != '1x' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;

**Word**

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Pg</td>
<td>Zn</td>
<td>Zd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UXTW**<sub>Zd</sub>.<sub>D</sub>, <sub>Pg</sub>/M, <sub>Zn</sub>.<sub>D</sub>

if !HaveSVE() then UNDEFINED;
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
Assembler Symbols

<Zd>  Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T>   For the byte variant: is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

For the halfword variant: is the size specifier, encoded in “size<0>”:

<table>
<thead>
<tr>
<th>size&lt;0&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

<Pg>   Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn>   Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    if Elem[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);
Z[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.


**UZP1, UZP2 (predicates)**

Concatenate even or odd elements from two predicates.

Concatenate adjacent even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: Even and Odd

**Even**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 1 0 1 | size | 1 0 | Pm | 0 1 0 0 1 0 0 | Pn | 0 | Pd |

**UZP1 <Pd>,<T>, <Pn><T>, <Pm><T>**

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer n = Uint(Pn);
integer m = Uint(Pm);
integer d = Uint(Pd);
integer part = 0;

**Odd**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 0 0 0 0 1 0 1 | size | 1 0 | Pm | 0 1 0 0 1 1 0 | Pn | 0 | Pd |

**UZP2 <Pd>,<T>, <Pn><T>, <Pm><T>**

if !HaveSVE() then UNDEFINED;
integer esize = 8 << Uint(size);
integer n = Uint(Pn);
integer m = Uint(Pm);
integer d = Uint(Pd);
integer part = 1;

**Assembler Symbols**

**<Pd>**  Is the name of the destination scalable predicate register, encoded in the "Pd" field.

**<T>**  Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

**<Pn>**  Is the name of the first source scalable predicate register, encoded in the "Pn" field.

**<Pm>**  Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

CheckSVEEnabled();
integer elements = \text{VL} \text{ DIV} \text{ esize};
bits(PL) operand1 = P[n];
bits(PL) operand2 = P[m];
bits(PL) result;

bits(PL+2) zipped = operand2:operand1;
for e = 0 to elements-1
    Elem[result, e, esize DIV 8] = Elem[zipped, 2*e+part, esize DIV 8];
P[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
UZP1, UZP2 (vectors)

Concatenate even or odd elements from two vectors.

Concatenate adjacent even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

It has encodings from 4 classes: Even, Even (quadwords), Odd and Odd (quadwords)

**Even**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 1 | size 1          | Zm 0 1 1 0 1 0 |
```

UZP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

**Even (quadwords)**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 1 | size 1          | Zm 0 0 0 0 1 0 0 |
```

UZP1 <Zd>.Q, <Zn>.Q, <Zm>.Q

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

**Odd**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 1 | size 1          | Zm 0 1 1 0 1 1 |
```

UZP2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

**Odd (quadwords)**

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 1 0 | size 1          | Zm 0 0 0 0 1 1 0 |
```
if !HaveSVEFP64MatMulExt() then UNDEFINED;

integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the ”Zd” field.

<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the ”Zn” field.

<Zm> Is the name of the second source scalable vector register, encoded in the ”Zm” field.

Operation

CheckSVEEnabled();
if VL < esize * 2 then UNDEFINED;
integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Zeros();

bits(VL*2) zipped = operand2:operand1;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];
Z[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WHILEGE

While decrementing signed scalar greater than or equal to scalar.

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element.

If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
0 0 1 0 0 1 0 1 | size | Rm | 0 0 0 | sf | 0 0 | Rn | 0 | Pd
```

WHILEGE <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_GE;
```

Assembler Symbols

- `<Pd>`: Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>`: Is the size specifier, encoded in "size":
  - size <T>
    - 00: B
    - 01: H
    - 10: S
    - 11: D
- `<R>`: Is a width specifier, encoded in "sf":
  - sf <R>
    - 0: W
    - 1: X
- `<n>`: Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
- `<m>`: Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
        last = last && cond;
        ElemP[result, e, esize] = if last then '1' else '0';
        operand1 = operand1 - 1;
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WHILEGT

While decrementing signed scalar greater than scalar.

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
| 0 0 1 0 0 1 0 1 | size | 1 | Rm | 0 0 0 | sf | 0 | 0 | Rn | 1 | Pd |

WHILEGT <Pd>..<T>, <R><n>, <R><m>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_GT;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in “sf”:

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rn” field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the “Rm” field.
**Operation**

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
    boolean cond;
    case op of
    when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
    when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
WHILEHI

While decrementing unsigned scalar higher than scalar:

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

### Assembler Symbols

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>` Is the size specifier, encoded in "size":
  
|
|---|---|
| size | `<T>` |
| 00 | B |
| 01 | H |
| 10 | S |
| 11 | D |

- `<R>` Is a width specifier, encoded in "sf":
  
|
|---|---|
| sf | `<R>` |
| 0 | W |
| 1 | X |

- `<n>` Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
- `<m>` Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

```bash
WHILEHI <Pd>.<T>, <R><n>, <R><m>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << Uint(size);
integer rsize = 32 << Uint(sf);
integer n = Uint(Rn);
integer m = Uint(Rm);
integer d = Uint(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_GT;
```
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
  boolean cond;
  case op of
    when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
    when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

  last = last && cond;
  ElemP[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WHILEHS

While decrementing unsigned scalar higher or same as scalar.

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element.

If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

WHILEHS <Pd>.<T>, <R><n>, <R><m>

```javascript
if !HaveSVE2() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_GE;
```

Assembler Symbols

| <Pd> | Is the name of the destination scalable predicate register, encoded in the "Pd" field. |
| <T>  | Is the size specifier, encoded in “size”: |
|      | size | <T> |
| 00   | B    |
| 01   | H    |
| 10   | S    |
| 11   | D    |

| <R>  | Is a width specifier, encoded in “sf”: |
|      | sf   | <R> |
| 0    | W    |
| 1    | X    |

| <n>  | Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field. |
| <m>  | Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field. |
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) > Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));
    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WHILELE

While incrementing signed scalar less than or equal to scalar.

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

### Assembler Symbols

- **<Pd>**
  - Is the name of the destination scalable predicate register, encoded in the "Pd" field.

- **<T>**
  - Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- **<R>**
  - Is a width specifier, encoded in "sf":

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

- **<n>**
  - Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

- **<m>**
  - Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;

for e = 0 to elements-1
  boolean cond;
  case op of
    when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
    when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
  last = last && cond;
  ElemP[result, e, esize] = if last then '1' else '0';
  operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WHILELO

While incrementing unsigned scalar lower than scalar.

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  |

WHILELO <Pd>,<T>, <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_LT;

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<R> Is a width specifier, encoded in "sf":

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n];
bits(rsize) operand2 = X[m];
bits(PL) result;
boolean last = TRUE;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WHILELS

While incrementing unsigned scalar lower or same as scalar.

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element.

If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.

The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

```
WHILELS <Pd>..<T>, <R><n>, <R><m>
``` 

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_LE;

Assembler Symbols

<Pa> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<RA> Is a width specifier, encoded in "sf":

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;RA&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.
Operation

```c
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(<PL>) mask = Ones(<PL>);
bits(rszie) operand1 = X[n];
bits(rszie) operand2 = X[m];
bits(<PL>) result;
boolean last = TRUE;

for e = 0 to elements - 1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) < Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));
    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WHILELT

While incrementing signed scalar less than scalar.

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element. The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

**Assembler Symbols**

- `<Pd>` Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- `<T>` Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- `<R>` Is a width specifier, encoded in "sf":

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

- `<n>` Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.
- `<m>` Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

```assembly
if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_LT;
```
Operation

CheckSVEEnabled();
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(\texttt{PL}) mask = \texttt{Ones(PL)};
bits(\texttt{rsize}) operand1 = \texttt{X}[n];
bits(\texttt{rsize}) operand2 = \texttt{X}[m];
bits(\texttt{PL}) result;
boolean last = \texttt{TRUE};

for e = 0 to elements-1
    boolean cond;
    case op of
        when \texttt{Cmp\_LT} cond = (\texttt{Int(operand1, unsigned)} < \texttt{Int(operand2, unsigned)});
        when \texttt{Cmp\_LE} cond = (\texttt{Int(operand1, unsigned)} \leq \texttt{Int(operand2, unsigned)});
    last = last && cond;
    \texttt{Elemp}\[\texttt{result, e, esize}\] = if last then '1' else '0';
    operand1 = operand1 + 1;

PSTATE.\langle N,Z,C,V \rangle = \texttt{PredTest}(mask, result, esize);
P[d] = result;

Operational information

If \texttt{PSTATE.DIT} is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
WHILERW

While free of read-after-write conflicts.

This instruction checks two addresses for a conflict or overlap between address ranges of the form \([addr, addr + VL / 8]\), where \(VL\) is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. Sets the \(FIRST\) (\(N\)), \(NONE\) (\(Z\)), \(!LAST\) (\(C\)) condition flags based on the predicate result, and the \(V\) flag to zero.

**Assembler Symbols**

\(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Xn>\) Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

\(<Xm>\) Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

**Operation**

\(\text{CheckSVEEnabled}();\)

integer elements = \(VL\) DIV esize;

bits\((PL)\) mask = \(\text{Ones}(PL)\);

bits(64) src1 = \(X[n]\);

bits(64) src2 = \(X[m]\);

integer operand1 = \(\text{UInt}(src1)\);

integer operand2 = \(\text{UInt}(src2)\);

bits\((PL)\) result;

integer diff = \(\text{Abs}(\text{operand2} - \text{operand1})\) DIV (esize DIV 8);

for e = 0 to elements - 1

\n
if diff == 0 || e < diff then

\(\text{ElemP}(\text{result}, e, \text{esize}) = '1';\)

else

\(\text{ElemP}(\text{result}, e, \text{esize}) = '0';\)

\n
\(\text{PSTATE}.<N,Z,C,V> = \text{PredTest}(\text{mask}, \text{result}, \text{esize});\)

\(P[d] = \text{result};\)
WHILEWR

While free of write-after-read/write conflicts.

This instruction checks two addresses for a conflict or overlap between address ranges of the form \( \text{addr,addr+VL÷8} \), where \( VL \) is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

Sets the \( \text{FIRST(N), NONE(Z), !LAST(C)} \) condition flags based on the predicate result, and the V flag to zero.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 size 1 Rm 0 0 1 1 0 0 Rn 0 0 0 0 Pd 0 rw

WHILEWR \(<Pd>,<T>,<Xn>,<Xm>\)

if \(!\text{HaveSVE2}()\) then UNDEFINED;
integer esize = 8 << \( \text{UInt(size)} \);
integer n = \( \text{UInt(Rn)} \);
integer m = \( \text{UInt(Rm)} \);
integer d = \( \text{UInt(Pd)} \);

Assembler Symbols

\(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\(<T>\) Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

\(<Xn>\) Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

\(<Xm>\) Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

Operation

\(\text{CheckSVEEnabled}()\);
integer elements = \( \text{VL} \) DIV esize;
bits(PL) mask = \( \text{Ones(PL)} \);
bits(64) src1 = \( X[n] \);
bits(64) src2 = \( X[m] \);
integer operand1 = \( \text{UInt(src1)} \);
integer operand2 = \( \text{UInt(src2)} \);
bits(PL) result;

integer diff = (operand2 - operand1) DIV (esize DIV 8);
for e = 0 to elements-1
    if diff <= 0 || e < diff then
        \( \text{ElemP[result, e, esize]} = '1'; \)
    else
        \( \text{ElemP[result, e, esize]} = '0'; \)

\(\text{PSTATE.<N,Z,C,V>} = \text{PredTest(mask, result, esize)};\)
\(\text{P[d]} = \text{result};\)
WRFFR

Write the first-fault register.

Read the source predicate register and place in the first-fault register (FFR). This instruction is intended to restore a saved FFR and is not recommended for general use by applications.

This instruction requires that the source predicate contains a MONOTONIC predicate value, in which starting from bit 0 there are zero or more 1 bits, followed only by 0 bits in any remaining bit positions. If the source is not a monotonic predicate value, then the resulting value in the FFR will be UNPREDICTABLE. It is not possible to generate a non-monotonic value in FFR when using SETFFR followed by first-fault or non-fault loads.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 0 0 0

WRFFR <Pn>.B

if !HaveSVE() then UNDEFINED;
integer n = UInt(Pn);

Assembler Symbols

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

CheckSVEEnabled();
bits(PL) operand = P[n];

hsb = HighestSetBit(operand);
if hsb < 0 || IsOnes(operand<hsb:0>) then
    FFR[] = operand;
else // not a monotonic predicate
    FFR[] = bits(PL) UNKNOWN;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
XAR

Bitwise exclusive OR and rotate right by immediate.

Bitwise exclusive OR the corresponding elements of the first and second source vectors, then rotate each result element right by an immediate amount. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-------------------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 0 0 1 0 0 tszh   | 1 tszl          | imm3            | 0 0 1 1 0 1     | Zm              | Zdn             |
```

XAR <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<const>

if !HaveSVE2() then UNDEFINED;
bits(4) tsize = tszh:tszl;
case tsize of
  when '0000' UNDEFINED;
  when '0001' esize = 8;
  when '001x' esize = 16;
  when '01xx' esize = 32;
  when '1xxx' esize = 64;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
integer rot = (2 * esize) - UInt(tsize:imm3);

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

<table>
<thead>
<tr>
<th>tszh</th>
<th>tszl</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>00</td>
<td>1x</td>
<td>H</td>
</tr>
<tr>
<td>01</td>
<td>xx</td>
<td>S</td>
</tr>
<tr>
<td>1x</td>
<td>xx</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.
<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in “tsz:imm3”.

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) result;
for e = 0 to elements-1
  bits(esize) element1 = Elem[operand1, e, esize];
  bits(esize) element2 = Elem[operand2, e, esize];
  Elem[result, e, esize] = ROR(element1 EOR element2, rot);
Z[dn] = result;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
ZIP1, ZIP2 (predicates)

Interleave elements from two half predicates.

Interleave alternating elements from the lowest or highest halves of the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.

It has encodings from 2 classes: High halves and Low halves

High halves

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size | 1 0 | Pm | 0 1 0 0 0 1 0 0 | Pn | 0 | Pd |
|-------------------------------|--------|-----|----|---|----------------|-----|---|-----|
| H                             |        |    |   |   |                 |     |   |     |

ZIP2 \(<Pd>.<T>, <Pn>.<T>, <Pm>.<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer n = \text{UInt}(\text{Pn});
integer m = \text{UInt}(\text{Pm});
integer d = \text{UInt}(\text{Pd});
integer part = 1;

Low halves

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size | 1 0 | Pm | 0 1 0 0 0 0 0 0 | Pn | 0 | Pd |
|-------------------------------|--------|-----|----|---|----------------|-----|---|-----|
| H                             |        |    |   |   |                 |     |   |     |

ZIP1 \(<Pd>.<T>, <Pn>.<T>, <Pm>.<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 << \text{UInt}(\text{size});
integer n = \text{UInt}(\text{Pn});
integer m = \text{UInt}(\text{Pm});
integer d = \text{UInt}(\text{Pd});
integer part = 0;

Assembler Symbols

- \(<Pd>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.
- \(<T>\) Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

- \(<Pn>\) Is the name of the first source scalable predicate register, encoded in the "Pn" field.
- \(<Pm>\) Is the name of the second source scalable predicate register, encoded in the "Pm" field.
Operation

`CheckSVEEnabled();`
integer pairs = `VL` DIV (esize * 2);
bits(PL) operand1 = `P[n];`
bits(PL) operand2 = `P[m];`
bits(PL) result;

integer base = part * pairs;
for p = 0 to pairs-1
    
    Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, base+p, esize DIV 8];
    Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, base+p, esize DIV 8];

P[d] = result;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
ZIP1, ZIP2 (vectors)

Interleave elements from two half vectors.

Interleave alternating elements from the lowest or highest halves of the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.

It has encodings from 4 classes: High halves, High halves (quadwords), Low halves and Low halves (quadwords)

High halves

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|--------|
| 0 0 0 0 0 1 0 1 | Zm 0 1 1 0 0 1 | Zn Zd |

ZIP2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

High halves (quadwords)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|--------|
| 0 0 0 0 0 1 0 1 | Zm 0 0 0 0 0 1 | Zn Zd |

ZIP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;

Low halves

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|--------|
| 0 0 0 0 0 1 0 1 | Zm 0 1 1 0 0 0 | Zn Zd |

ZIP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Low halves (quadwords)

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---------------|---------------|--------|
| 0 0 0 0 0 1 0 1 | Zm 0 0 0 0 0 0 | Zn Zd |
if !HaveSVEFP64MatMulExt() then UNDEFINED;
integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.
<T> Is the size specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
</tr>
</tbody>
</table>

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

CheckSVEEnabled();
if VL < esize * 2 then UNDEFINED;
integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Zeros();

integer base = part * pairs;
for p = 0 to pairs-1
   \[ \text{Elem}[result, 2*p+0, esize] = \text{Elem}[operand1, base+p, esize]; \]
   \[ \text{Elem}[result, 2*p+1, esize] = \text{Elem}[operand2, base+p, esize]; \]
\[ Z[d] = \text{result}; \]

Operational information

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.

• The response of this instruction to asynchronous exceptions does not vary based on:
  • The values of the data supplied in any of its registers.
  • The values of the NZCV flags.
Top-level encodings for A64

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>SVE encodings</td>
<td></td>
</tr>
<tr>
<td>0011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>100x</td>
<td>Data Processing -- Immediate</td>
<td></td>
</tr>
<tr>
<td>101x</td>
<td>Branches, Exception Generating and System instructions</td>
<td></td>
</tr>
<tr>
<td>x1x0</td>
<td>Loads and Stores</td>
<td></td>
</tr>
<tr>
<td>x101</td>
<td>Data Processing -- Register</td>
<td></td>
</tr>
<tr>
<td>x111</td>
<td>Data Processing -- Scalar Floating-Point and Advanced SIMD</td>
<td></td>
</tr>
</tbody>
</table>

Reserved

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>UDF</td>
<td></td>
</tr>
<tr>
<td>!= 00000000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

SVE encodings

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>x1xxxx</td>
<td>SVE Integer Multiply-Add - Predicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>000xxxx</td>
<td>SVE Integer Binary Arithmetic - Predicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>01xxxx</td>
<td>SVE Integer Reduction</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>100xxxx</td>
<td>SVE Bitwise Shift - Predicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>101xxxx</td>
<td>SVE Integer Unary Arithmetic - Predicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>000xxxx</td>
<td>SVE integer add/subtract vectors (unpredicated)</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>001xxxx</td>
<td>SVE Bitwise Logical - Unpredicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>0100xx</td>
<td>SVE Index Generation</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>0101xx</td>
<td>SVE Stack Allocation</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>011xxxx</td>
<td>SVE2 Integer Multiply - Unpredicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>100xxxx</td>
<td>SVE Bitwise Shift - Unpredicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>1010xx</td>
<td>SVE address generation</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>1011xx</td>
<td>SVE Integer Misc - Unpredicated</td>
</tr>
<tr>
<td>Format</td>
<td>Opcode</td>
<td>Instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxxx   11xxxxx</td>
<td>SVE Element Count</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>00xxx</td>
<td>SVE Bitwise Immediate</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>01xxx</td>
<td>SVE Integer Wide Immediate - Predicated</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   001000</td>
<td>DUP (indexed)</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   001001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   00101x</td>
<td>SVE table lookup (three sources)</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   0011x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   001100</td>
<td>TBL — SVE</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   001110</td>
<td>SVE Permute Vector - Unpredicated</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   010xxx</td>
<td>SVE Permute Predicate</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   011xxx</td>
<td>SVE permute vector elements</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   10xxxx</td>
<td>SVE Permute Vector - Predicated</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x</td>
<td>1xxxxx   11xxxx</td>
<td>SEL (vectors)</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>10</td>
<td>1xxxxx   000xxx</td>
<td>SVE Permute Vector - Extract</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0x</td>
<td>0xxxxx</td>
<td>SVE permute vector segments</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>0x</td>
<td>1xxxxx</td>
<td>SVE integer compare with unsigned immediate</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>0xxxxx   x0xxxx</td>
<td>SVE integer compare with signed immediate</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>0xxxxx   01xxxx</td>
<td>SVE predicate logical operations</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>0xxxxx   11xxxx</td>
<td>SVE Propagate Break</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>01xxx   01xxxx</td>
<td>SVE Partition Break</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>01xxx   11xxxx</td>
<td>SVE Predicate Misc</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxxxx   00xxxx</td>
<td>SVE Integer Compare - Scalars</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxxxx   01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>1xxxxx   11xxxx</td>
<td>SVE Integer Wide Immediate - Unpredicated</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>100xx   10xxxx</td>
<td>SVE predicate count</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx   1000xx</td>
<td>SVE Inc/Dec by Predicate Count</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx   1001xx</td>
<td>SVE Write FFR</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>1x</td>
<td>101xx   101xxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0x</td>
<td>0xxxxx   0xxxxx</td>
<td>SVE Integer Multiply-Add - Unpredicated</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0x</td>
<td>0xxxxx   10xxxx</td>
<td>SVE2 Integer - Predicated</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0x</td>
<td>0xxxxx   11xxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>0x</td>
<td>1xxxxx</td>
<td>SVE Multiply - Indexed</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>0xxxxx   0xxxxx</td>
<td>SVE2 Widening Integer Arithmetic</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>0xxxxx   10xxxx</td>
<td>SVE Misc</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>0xxxxx   11xxxx</td>
<td>SVE2 Accumulate</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>1xxxxx   0xxxxx</td>
<td>SVE2 Narrowing</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>1xxxxx   100xxx</td>
<td>SVE2 character match</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>1xxxxx   101xxx</td>
<td>SVE2 Histogram Computation - Segment</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>1xxxxx   110xxx</td>
<td>HISTCNT</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>1x</td>
<td>1xxxxx   111xxx</td>
<td>SVE2 Crypto Extensions</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>0xxxxx   0xxxxx</td>
<td>FCMLA (vectors)</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>001x1x</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>000000</td>
<td>100xxx</td>
<td>FCADD</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>00000</td>
<td>101xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>00000</td>
<td>11xxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>00001</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>0x</td>
<td>0010x</td>
<td>100xxx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE Integer Multiply-Add - Predicated

These instructions are under SVE encodings.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 00000100 | 0 | op0 | 1 |

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \theta )</td>
<td>SVE integer multiply-accumulate writing addend (predicated)</td>
</tr>
</tbody>
</table>
SVE integer multiply-add writing multiplicand (predicated)

These instructions are under SVE Integer Multiply-Add - Predicated.

```
0 0 0 0 0 0 0 0 1 0 0 | size 0 | Zm 0 1 op | Pg | Zn | Zda
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MLA (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>MLS (vectors)</td>
</tr>
</tbody>
</table>

SVE integer multiply-accumulate writing addend (predicated)

These instructions are under SVE Integer Multiply-Add - Predicated.

```
0 0 0 0 0 0 0 0 1 0 0 | size 0 | Zm 1 1 op | Pg | Za | Zdn
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MAD</td>
</tr>
<tr>
<td>1</td>
<td>MSB</td>
</tr>
</tbody>
</table>

SVE Integer Binary Arithmetic - Predicated

These instructions are under SVE encodings.

```
00000100 0 0 | op0 | 000
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000x</td>
<td>SVE integer add/subtract vectors (predicated)</td>
</tr>
<tr>
<td>01x</td>
<td>SVE integer min/max/difference (predicated)</td>
</tr>
<tr>
<td>100</td>
<td>SVE integer multiply vectors (predicated)</td>
</tr>
<tr>
<td>101</td>
<td>SVE integer divide vectors (predicated)</td>
</tr>
<tr>
<td>11x</td>
<td>SVE bitwise logical operations (predicated)</td>
</tr>
</tbody>
</table>

SVE integer add/subtract vectors (predicated)

These instructions are under SVE Integer Binary Arithmetic - Predicated.

```
0 0 0 0 0 0 0 1 0 0 | size 0 0 0 | opc 0 0 0 | Pg | Zm | Zdn
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD (vectors, predicated)</td>
</tr>
<tr>
<td>001</td>
<td>SUB (vectors, predicated)</td>
</tr>
<tr>
<td>010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011</td>
<td>SUBR (vectors)</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
**SVE integer min/max/difference (predicated)**

These instructions are under **SVE Integer Binary Arithmetic - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc U</td>
<td></td>
</tr>
<tr>
<td>00 0</td>
<td>SMAX (vectors)</td>
</tr>
<tr>
<td>00 1</td>
<td>UMAX (vectors)</td>
</tr>
<tr>
<td>01 0</td>
<td>SMIN (vectors)</td>
</tr>
<tr>
<td>01 1</td>
<td>UMIN (vectors)</td>
</tr>
<tr>
<td>10 0</td>
<td>SABD</td>
</tr>
<tr>
<td>10 1</td>
<td>UABD</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE integer multiply vectors (predicated)**

These instructions are under **SVE Integer Binary Arithmetic - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc U</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>MUL (vectors, predicated)</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0</td>
<td>SMULH (predicated)</td>
</tr>
<tr>
<td>1 1</td>
<td>UMULH (predicated)</td>
</tr>
</tbody>
</table>

**SVE integer divide vectors (predicated)**

These instructions are under **SVE Integer Binary Arithmetic - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc U</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>SDIV</td>
</tr>
<tr>
<td>0 1</td>
<td>UDIV</td>
</tr>
<tr>
<td>1 0</td>
<td>SDIVR</td>
</tr>
<tr>
<td>1 1</td>
<td>UDIVR</td>
</tr>
</tbody>
</table>

**SVE bitwise logical operations (predicated)**

These instructions are under **SVE Integer Binary Arithmetic - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>ORR (vectors, predicated)</td>
</tr>
<tr>
<td>001</td>
<td>EOR (vectors, predicated)</td>
</tr>
</tbody>
</table>
SVE Integer Reduction

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>AND (vectors, predicated)</td>
</tr>
<tr>
<td>011</td>
<td>BIC (vectors, predicated)</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE integer add reduction (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE integer add reduction (predicated)</td>
</tr>
<tr>
<td>01</td>
<td>SVE integer min/max reduction (predicated)</td>
</tr>
<tr>
<td>10</td>
<td>SVE constructive prefix (predicated)</td>
</tr>
<tr>
<td>11</td>
<td>SVE bitwise logical reduction (predicated)</td>
</tr>
</tbody>
</table>

SVE integer min/max reduction (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SADDV</td>
</tr>
<tr>
<td>01</td>
<td>UADDV</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE constructive prefix (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SMAXV</td>
</tr>
<tr>
<td>01</td>
<td>UMAXV</td>
</tr>
<tr>
<td>01</td>
<td>SMINV</td>
</tr>
<tr>
<td>1x</td>
<td>UMINV</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### SVE bitwise logical reduction (predicated)

These instructions are under [SVE Integer Reduction](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>MOVPRFX (predicated)</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE Bitwise Shift - Predicated

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ORV</td>
</tr>
<tr>
<td>001</td>
<td>EORV</td>
</tr>
<tr>
<td>010</td>
<td>ANDV</td>
</tr>
<tr>
<td>011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE bitwise shift by immediate (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>SVE bitwise shift by immediate (predicated)</td>
</tr>
<tr>
<td>10</td>
<td>SVE bitwise shift by vector (predicated)</td>
</tr>
<tr>
<td>11</td>
<td>SVE bitwise shift by wide elements (predicated)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ASR (immediate, predicated)</td>
</tr>
<tr>
<td>00</td>
<td>LSR (immediate, predicated)</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>LSL (immediate, predicated)</td>
</tr>
<tr>
<td>01</td>
<td>ASRD</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>SQSHL (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>UQSHL (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>SRSHR</td>
</tr>
<tr>
<td>11</td>
<td>URSHR</td>
</tr>
</tbody>
</table>
SVE bitwise shift by vector (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 1 0 0 | size | 0 1 0 | R | L | U | 1 0 0 | Pg | Zm | Zdn
```

SVE bitwise shift by wide elements (predicated)

These instructions are under [SVE Bitwise Shift - Predicated](#).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 1 0 0 | size | 0 1 1 | R | L | U | 1 0 0 | Pg | Zm | Zdn
```

SVE Integer Unary Arithmetic - Predicated

These instructions are under [SVE encodings](#).

```
00000100 0 | op0 | 101
```

### SVE integer unary operations (predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).
### SVE Bitwise Unary Operations (Predicated)

These instructions are under [SVE Integer Unary Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>SXTB, SXTH, SXTW — SXTB</td>
</tr>
<tr>
<td>001</td>
<td>UXTB, UXTH, UXTW — UXTB</td>
</tr>
<tr>
<td>010</td>
<td>SXTB, SXTH, SXTW — SXTH</td>
</tr>
<tr>
<td>011</td>
<td>UXTB, UXTH, UXTW — UXTH</td>
</tr>
<tr>
<td>100</td>
<td>SXTB, SXTH, SXTW — SXTW</td>
</tr>
<tr>
<td>101</td>
<td>UXTB, UXTH, UXTW — UXTW</td>
</tr>
<tr>
<td>110</td>
<td>ABS</td>
</tr>
<tr>
<td>111</td>
<td>NEG</td>
</tr>
</tbody>
</table>

### SVE Integer Add/Subtract Vectors (Unpredicated)

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ADD (vectors, unpredicated)</td>
</tr>
<tr>
<td>001</td>
<td>SUB (vectors, unpredicated)</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100</td>
<td>SQADD (vectors, unpredicated)</td>
</tr>
<tr>
<td>101</td>
<td>UQADD (vectors, unpredicated)</td>
</tr>
<tr>
<td>110</td>
<td>SQSUB (vectors, unpredicated)</td>
</tr>
<tr>
<td>111</td>
<td>UQSUB (vectors, unpredicated)</td>
</tr>
</tbody>
</table>

### SVE bitwise logical - Unpredicated

These instructions are under [SVE encodings](#).
### SVE bitwise logical operations (unpredicated)

These instructions are under **SVE Bitwise Logical - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100</td>
<td>SVE bitwise logical operations (unpredicated)</td>
</tr>
<tr>
<td>101</td>
<td>XAR</td>
</tr>
<tr>
<td>11x</td>
<td>SVE2 bitwise ternary operations</td>
</tr>
</tbody>
</table>

### SVE bitwise ternary operations

These instructions are under **SVE Bitwise Logical - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>AND (vectors, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>ORR (vectors, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>EOR (vectors, unpredicated)</td>
</tr>
<tr>
<td>11</td>
<td>BIC (vectors, unpredicated)</td>
</tr>
</tbody>
</table>

### SVE Index Generation

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000100</td>
<td>INDEX (immediates)</td>
</tr>
<tr>
<td>0100</td>
<td>op0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>INDEX (immediates)</td>
</tr>
<tr>
<td>01</td>
<td>INDEX (scalar, immediate)</td>
</tr>
<tr>
<td>10</td>
<td>INDEX (immediate, scalar)</td>
</tr>
<tr>
<td>11</td>
<td>INDEX (scalars)</td>
</tr>
</tbody>
</table>
SVE Stack Allocation

These instructions are under SVE encodings.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
00000100 | op0 | 1 |
0101 | op1 |

Decode fields
op0 op1 Instruction details
0 0 SVE stack frame adjustment
1 0 SVE stack frame size
1 UNALLOCATED

SVE stack frame adjustment

These instructions are under SVE Stack Allocation.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 0 | op | 1 |
Rn 0 1 0 1 0 | imm6 | Rd

Decode fields
op Instruction Details
0 ADDVL
1 ADDPL

SVE stack frame size

These instructions are under SVE Stack Allocation.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 1 0 0 1 | op | 1 |
opc2 0 1 0 1 0 | imm6 | Rd

Decode fields
op opc2 Instruction Details
0 0xxxx UNALLOCATED
0 10xxxx UNALLOCATED
0 110xx UNALLOCATED
0 1110x UNALLOCATED
0 11110 UNALLOCATED
0 11111 RDVL
1 UNALLOCATED

SVE2 Integer Multiply - Unpredicated

These instructions are under SVE encodings.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
00000100 | 1 |
011 | op0 |

Decode fields
op0 Instruction details
0x SVE2 integer multiply vectors (unpredicated)
10 SVE2 signed saturating doubling multiply high (unpredicated)
11 UNALLOCATED
## SVE2 Integer Multiply Vectors (unpredicated)

These instructions are under [SVE2 Integer Multiply - Unpredicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>MUL (vectors, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>SMULH (unpredicated)</td>
</tr>
<tr>
<td>11</td>
<td>UMULH (unpredicated)</td>
</tr>
<tr>
<td>00 01</td>
<td>PMUL</td>
</tr>
<tr>
<td>01 01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x 01</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

## SVE2 Signed Saturating Doubling Multiply High (unpredicated)

These instructions are under [SVE2 Integer Multiply - Unpredicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SQDMULH (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>SQRDMULH (vectors)</td>
</tr>
</tbody>
</table>

## SVE Bitwise Shift - Unpredicated

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE bitwise shift by wide elements (unpredicated)</td>
</tr>
<tr>
<td>1</td>
<td>SVE bitwise shift by immediate (unpredicated)</td>
</tr>
</tbody>
</table>

## SVE Bitwise Shift by Wide Elements (unpredicated)

These instructions are under [SVE Bitwise Shift - Unpredicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ASR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>LSR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LSL (wide elements, unpredicated)</td>
</tr>
</tbody>
</table>

## SVE Bitwise Shift by Immediate (unpredicated)

These instructions are under [SVE Bitwise Shift - Unpredicated](#).
### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ASR (immediate, unpredicated)</td>
</tr>
<tr>
<td>01</td>
<td>LSR (immediate, unpredicated)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>LSL (immediate, unpredicated)</td>
</tr>
</tbody>
</table>

### SVE address generation

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ADR — Unpacked 32-bit signed offsets</td>
</tr>
<tr>
<td>01</td>
<td>ADR — Unpacked 32-bit unsigned offsets</td>
</tr>
<tr>
<td>1x</td>
<td>ADR — Packed offsets</td>
</tr>
</tbody>
</table>

### SVE Integer Misc - Unpredicated

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>SVE floating-point trig select coefficient</td>
</tr>
<tr>
<td>10</td>
<td>SVE floating-point exponential accelerator</td>
</tr>
<tr>
<td>11</td>
<td>SVE constructive prefix (unpredicated)</td>
</tr>
</tbody>
</table>

### SVE floating-point trig select coefficient

These instructions are under **SVE Integer Misc - Unpredicated**.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FTSSEL</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE floating-point exponential accelerator

These instructions are under **SVE Integer Misc - Unpredicated**.
### SVE constructive prefix (unpredicated)

These instructions are under **SVE Integer Misc - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>FEXPA</td>
</tr>
<tr>
<td>00001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xxxx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE Element Count

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>MOVPRFX (unpredicated)</td>
</tr>
<tr>
<td>00 00000</td>
<td></td>
</tr>
<tr>
<td>00 00001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 001xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 01xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 1xxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE saturating inc/dec vector by element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields op1</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE saturating inc/dec vector by element count</td>
</tr>
<tr>
<td>0100</td>
<td>SVE element count</td>
</tr>
<tr>
<td>0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000</td>
<td>SVE inc/dec vector by element count</td>
</tr>
<tr>
<td>100</td>
<td>SVE inc/dec register by element count</td>
</tr>
<tr>
<td>1x01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11x</td>
<td>SVE saturating inc/dec register by element count</td>
</tr>
</tbody>
</table>

### SVE saturating inc/dec vector by element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 00x</td>
<td>SVE saturating inc/dec vector by element count</td>
</tr>
<tr>
<td>0 100</td>
<td>SVE element count</td>
</tr>
<tr>
<td>0 101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 000</td>
<td>SVE inc/dec vector by element count</td>
</tr>
<tr>
<td>1 100</td>
<td>SVE inc/dec register by element count</td>
</tr>
<tr>
<td>1 x01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11x</td>
<td>SVE saturating inc/dec register by element count</td>
</tr>
</tbody>
</table>
### SVE element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields size</th>
<th>D</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 0 0</td>
<td></td>
<td></td>
<td>SQINCH (vector)</td>
</tr>
<tr>
<td>01 0 1</td>
<td></td>
<td></td>
<td>UQINCH (vector)</td>
</tr>
<tr>
<td>01 1 0</td>
<td></td>
<td></td>
<td>SODECH (vector)</td>
</tr>
<tr>
<td>01 1 1</td>
<td></td>
<td></td>
<td>UODECH (vector)</td>
</tr>
<tr>
<td>10 0 0</td>
<td></td>
<td></td>
<td>SQINCW (vector)</td>
</tr>
<tr>
<td>10 0 1</td>
<td></td>
<td></td>
<td>UQINCW (vector)</td>
</tr>
<tr>
<td>10 1 0</td>
<td></td>
<td></td>
<td>SQDECW (vector)</td>
</tr>
<tr>
<td>10 1 1</td>
<td></td>
<td></td>
<td>UQDECW (vector)</td>
</tr>
<tr>
<td>11 0 0</td>
<td></td>
<td></td>
<td>SQINCD (vector)</td>
</tr>
<tr>
<td>11 0 1</td>
<td></td>
<td></td>
<td>UQINCD (vector)</td>
</tr>
<tr>
<td>11 1 0</td>
<td></td>
<td></td>
<td>SQDECD (vector)</td>
</tr>
<tr>
<td>11 1 1</td>
<td></td>
<td></td>
<td>UQDECD (vector)</td>
</tr>
</tbody>
</table>

### SVE inc/dec vector by element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields size</th>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 0</td>
<td></td>
<td>CNTB, CNTD, CNTH, CNTW — CNTB</td>
</tr>
<tr>
<td>10 0</td>
<td></td>
<td>CNTB, CNTD, CNTH, CNTW — CNTH</td>
</tr>
<tr>
<td>11 0</td>
<td></td>
<td>CNTB, CNTD, CNTH, CNTW — CNTD</td>
</tr>
</tbody>
</table>

### SVE inc/dec register by element count

These instructions are under **SVE Element Count**.

<table>
<thead>
<tr>
<th>Decode fields size</th>
<th>D</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 0</td>
<td></td>
<td>INCD, INCH, INCW (vector) — INCH</td>
</tr>
<tr>
<td>01 1</td>
<td></td>
<td>DECD, DECH, DECW (vector) — DECH</td>
</tr>
<tr>
<td>10 0</td>
<td></td>
<td>INCD, INCH, INCW (vector) — INCW</td>
</tr>
<tr>
<td>10 1</td>
<td></td>
<td>DECD, DECH, DECW (vector) — DECW</td>
</tr>
<tr>
<td>11 0</td>
<td></td>
<td>INCD, INCH, INCW (vector) — INCD</td>
</tr>
<tr>
<td>11 1</td>
<td></td>
<td>DECD, DECH, DECW (vector) — DECD</td>
</tr>
</tbody>
</table>
SVE saturating inc/dec register by element count

These instructions are under SVE Element Count.
SVE Bitwise Immediate

These instructions are under SVE encodings.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|---------------|---------------|
| 000000101 | op0 | 00 | op1 |
```

**Decode fields**

| 11 | 00 | DUPM |
| != 11 | 00 | SVE bitwise logical with immediate (unpredicated) |
| != 00 | UNALLOCATED |

SVE bitwise logical with immediate (unpredicated)

These instructions are under SVE Bitwise Immediate.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|---------------|---------------|
| 0 0 0 0 0 1 0 1 | != 11 | 0 0 0 0 | imm13 |
| opc |
```

The following constraints also apply to this encoding: opc != 11 && opc != 11

**Decode fields**

| 00 | ORR (immediate) |
| 01 | EOR (immediate) |
| 10 | AND (immediate) |

SVE Integer Wide Immediate - Predicated

These instructions are under SVE encodings.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|---------------|---------------|
| 000000101 | 01 | op0 |
```

**Decode fields**

| 0xx | SVE copy integer immediate (predicated) |
| 10x | UNALLOCATED |
| 110 | FCPY |
| 111 | UNALLOCATED |

SVE copy integer immediate (predicated)

These instructions are under SVE Integer Wide Immediate - Predicated.

```
| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 |
|-----------------------------------------------|---------------|---------------|
| 0 0 0 0 0 1 0 1 | size | 0 1 | Pg | 0 | M | sh | imm8 | Zd |
```
SVE table lookup (three sources)

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>CPY (immediate, zeroing)</td>
</tr>
<tr>
<td></td>
<td>CPY (immediate, merging)</td>
</tr>
</tbody>
</table>

SVE Permute Vector - Unpredicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>TBL</td>
</tr>
<tr>
<td></td>
<td>TBX</td>
</tr>
</tbody>
</table>

SVE unpack vector elements

These instructions are under SVE Permute Vector - Unpredicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>DUP (scalar)</td>
</tr>
<tr>
<td></td>
<td>INSR (scalar)</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>SVE unpack vector elements</td>
</tr>
<tr>
<td></td>
<td>INSR (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>REV (vector)</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE unpack vector elements

These instructions are under SVE Permute Vector - Unpredicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U H</td>
<td>SUNPKHI, SUNPKLO — SUNPKLO</td>
</tr>
<tr>
<td></td>
<td>SUNPKHI, SUNPKLO — SUNPKHI</td>
</tr>
<tr>
<td></td>
<td>UUNPKHI, UUNPKLO — UUNPKLO</td>
</tr>
<tr>
<td></td>
<td>UUNPKHI, UUNPKLO — UUNPKHI</td>
</tr>
</tbody>
</table>
SVE Permute Predicate

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1000x</td>
<td>0000</td>
<td>0</td>
<td>SVE unpack predicate elements</td>
</tr>
<tr>
<td>01</td>
<td>1000x</td>
<td>0000</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>1000x</td>
<td>0000</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1000x</td>
<td>0000</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0xxxx</td>
<td>xxxx0</td>
<td>0</td>
<td>SVE permute predicate elements</td>
<td></td>
</tr>
<tr>
<td>0xxxx</td>
<td>xxxx1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td>0000</td>
<td>0</td>
<td>REV (predicate)</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>0000</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x0x</td>
<td>1000</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x0x</td>
<td>x100</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x0x</td>
<td>xx10</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x0x</td>
<td>xxx1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>10x1x</td>
<td></td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>11xxx</td>
<td></td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

SVE unpack predicate elements

These instructions are under SVE Permute Predicate.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PUNPKHI, PUNPKLO — PUNPKLO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PUNPKHI, PUNPKLO — PUNPKHI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE permute predicate elements

These instructions are under SVE Permute Predicate.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ZIP1, ZIP2 (predicates) — ZIP1</td>
</tr>
<tr>
<td>00</td>
<td>ZIP1, ZIP2 (predicates) — ZIP2</td>
</tr>
<tr>
<td>01</td>
<td>UZP1, UZP2 (predicates) — UZP1</td>
</tr>
<tr>
<td>01</td>
<td>UZP1, UZP2 (predicates) — UZP2</td>
</tr>
<tr>
<td>10</td>
<td>TRN1, TRN2 (predicates) — TRN1</td>
</tr>
<tr>
<td>10</td>
<td>TRN1, TRN2 (predicates) — TRN2</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE permute vector elements

These instructions are under **SVE encodings**.

```
<table>
<thead>
<tr>
<th>size</th>
<th>Zm</th>
<th>opc</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ZIP1, ZIP2 (vectors) — ZIP1</td>
</tr>
<tr>
<td>001</td>
<td>ZIP1, ZIP2 (vectors) — ZIP2</td>
</tr>
<tr>
<td>010</td>
<td>UZP1, UZP2 (vectors) — UZP1</td>
</tr>
<tr>
<td>011</td>
<td>UZP1, UZP2 (vectors) — UZP2</td>
</tr>
<tr>
<td>100</td>
<td>TRN1, TRN2 (vectors) — TRN1</td>
</tr>
<tr>
<td>101</td>
<td>TRN1, TRN2 (vectors) — TRN2</td>
</tr>
<tr>
<td>11x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE Permute Vector - Predicated

These instructions are under **SVE encodings**.

```
<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode fields</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CPY (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>COMPACT</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SVE extract element to general register</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SVE extract element to SIMD&amp;FP scalar register</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SVE reverse within elements</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CPY (scalar)</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SVE conditionally broadcast element to vector</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SVE conditionally extract element to SIMD&amp;FP scalar</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SPLICE — Destructive</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SPLICE — Constructor</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SVE conditionally broadcast element to vector</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SVE conditionally extract element to general register</td>
</tr>
<tr>
<td>1</td>
<td>!= 000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE extract element to general register

These instructions are under **SVE Permute Vector - Predicated**.

```
<table>
<thead>
<tr>
<th>size</th>
<th>B</th>
<th>opc</th>
<th>Pg</th>
<th>Zn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

---

Page 2864
SVE extract element to SIMD&FP scalar register

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LASTA (scalar)</td>
</tr>
<tr>
<td>1</td>
<td>LASTB (scalar)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>opc</th>
<th>Pg</th>
<th>Zn</th>
<th>Vd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

SVE reverse within elements

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LASTA (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>1</td>
<td>LASTB (SIMD&amp;FP scalar)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>opc</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

SVE conditionally broadcast element to vector

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>REVB, REVH, REVW — REVB</td>
</tr>
<tr>
<td>1</td>
<td>REVB, REVH, REVW — REVH</td>
</tr>
<tr>
<td>10</td>
<td>REVB, REVH, REVW — REVW</td>
</tr>
<tr>
<td>11</td>
<td>RBIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>opc</th>
<th>Pg</th>
<th>Zn</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

SVE conditionally extract element to SIMD&FP scalar

These instructions are under SVE Permute Vector - Predicated.

<table>
<thead>
<tr>
<th>Decode fields B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLASTA (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (vectors)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Size</th>
<th>opc</th>
<th>Pg</th>
<th>Zn</th>
<th>Vdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
SVE conditionally extract element to general register

These instructions are under SVE Permute Vector - Predicated.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | size 1 1 0 0 0 | B 1 0 1 | Pg | Zm | Rdn |

Decode fields

<table>
<thead>
<tr>
<th>B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLASTA (scalar)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (scalar)</td>
</tr>
</tbody>
</table>

SVE Permute Vector - Extract

These instructions are under SVE encodings.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 000001010 | op0 | 1 | 000 |

Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>EXT — Destructive</td>
</tr>
<tr>
<td>1</td>
<td>EXT — Constructive</td>
</tr>
</tbody>
</table>

SVE permute vector segments

These instructions are under SVE encodings.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 0 0 0 0 0 1 0 1 1 | op 1 | Zm | 0 0 0 | opc2 | Zn | Zd |

Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>ZIP1, ZIP2 (vectors) — ZIP1</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>ZIP1, ZIP2 (vectors) — ZIP2</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>UZP1, UZP2 (vectors) — UZP1</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>UZP1, UZP2 (vectors) — UZP2</td>
</tr>
<tr>
<td>0</td>
<td>10x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>TRN1, TRN2 (vectors) — TRN1</td>
</tr>
<tr>
<td>0</td>
<td>111</td>
<td>TRN1, TRN2 (vectors) — TRN2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Integer Compare - Vectors

These instructions are under SVE encodings.

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | 00100100 | op0 |

Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE integer compare vectors</td>
</tr>
<tr>
<td>1</td>
<td>SVE integer compare with wide elements</td>
</tr>
</tbody>
</table>
SVE integer compare vectors

These instructions are under SVE Integer Compare - Vectors.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPHS</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPHI</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPEQ</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPNE</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPGE</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPGT</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPEQ</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPNE</td>
</tr>
</tbody>
</table>

SVE integer compare with wide elements

These instructions are under SVE Integer Compare - Vectors.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPGE</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPGT</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLT</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLE</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPHS</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPHI</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLQ</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLS</td>
</tr>
</tbody>
</table>

SVE integer compare with unsigned immediate

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPSH</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPSH</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPSH</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPSH</td>
</tr>
</tbody>
</table>

SVE integer compare with signed immediate

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPSH</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPSH</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPSH</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPSH</td>
</tr>
</tbody>
</table>
### SVE predicate logical operations

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op o2</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

### SVE Propagate Break

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op S o2 o3</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

### SVE Propagate Break from previous partition

These instructions are under **SVE Propagate Break**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
### SVE Partition Break

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>0</td>
<td>0</td>
<td><strong>SVE propagate break to next partition</strong></td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x000</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x1xx</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xx1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xxx1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td>0</td>
<td></td>
<td><strong>SVE partition break condition</strong></td>
</tr>
</tbody>
</table>

#### SVE propagate break to next partition

These instructions are under **SVE Partition Break**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td><strong>SVE propagate break to next partition</strong></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td><strong>SVE propagate break to next partition</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td><strong>SVE propagate break to next partition</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td><strong>SVE propagate break to next partition</strong></td>
</tr>
</tbody>
</table>

#### SVE partition break condition

These instructions are under **SVE Partition Break**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td><strong>SVE partition break condition</strong></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td><strong>SVE partition break condition</strong></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td><strong>SVE partition break condition</strong></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td><strong>SVE partition break condition</strong></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td><strong>SVE partition break condition</strong></td>
</tr>
</tbody>
</table>
### SVE Predicate Misc

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x0</td>
<td>0</td>
<td>SVE predicate test</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0xx1</td>
<td>x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0xxx</td>
<td>x1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>000</td>
<td>0</td>
<td>SVE predicate first active</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>000</td>
<td>!= 00</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>0</td>
<td>SVE predicate zero</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>!= 0000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>110</td>
<td>0</td>
<td>SVE predicate read from FFR (predicated)</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>000</td>
<td>0x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>000</td>
<td>10</td>
<td>PNEXT</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>000</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>100</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>110</td>
<td>00</td>
<td>SVE predicate read from FFR (unpredicated)</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>110</td>
<td>!= 0000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>100x</td>
<td>010</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>100x</td>
<td>100</td>
<td>0x</td>
<td>SVE predicate initialize</td>
<td></td>
</tr>
<tr>
<td>100x</td>
<td>100</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>100x</td>
<td>110</td>
<td>!= 00</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>100x</td>
<td>x1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>110x</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1x1x</td>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### SVE predicate test

These instructions are under SVE Predicate Misc.

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0000</td>
<td>PTEST</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### SVE predicate first active

These instructions are under **SVE Predicate Misc**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op S opc2</td>
<td></td>
</tr>
<tr>
<td>0 1 1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate zero

These instructions are under **SVE Predicate Misc**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op S</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1</td>
<td>PFIRST</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate read from FFR (predicated)

These instructions are under **SVE Predicate Misc**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op S</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>RDFFR, RDFFRS (predicated) — not setting the condition flags</td>
</tr>
<tr>
<td>0 1</td>
<td>RDFFR, RDFFRS (predicated) — setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate read from FFR (unpredicated)

These instructions are under **SVE Predicate Misc**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op S</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>RDFFR (unpredicated)</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE predicate initialize

These instructions are under SVE Predicate Misc.

<table>
<thead>
<tr>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PTRUE, PTRUES — not setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td>PTRUE, PTRUES — setting the condition flags</td>
</tr>
</tbody>
</table>

SVE Integer Compare - Scalars

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>00</td>
<td>0000</td>
<td>SVE integer compare scalar count and limit</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>!= 0000</td>
<td>SVE conditionally terminate scalars</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>!= 0000</td>
<td>SVE pointer conflict compare</td>
</tr>
<tr>
<td>1x</td>
<td>!= 00</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

SVE integer compare scalar count and limit

These instructions are under SVE Integer Compare - Scalars.

<table>
<thead>
<tr>
<th>U</th>
<th>lt</th>
<th>eq</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>WHILEGE</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>WHILEGT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>WHILELT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>WHILELE</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>WHILEHS</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>WHILEHI</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>WHILELO</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>WHILELS</td>
</tr>
</tbody>
</table>

SVE conditionally terminate scalars

These instructions are under SVE Integer Compare - Scalars.

<table>
<thead>
<tr>
<th>op</th>
<th>sz</th>
<th>Rm</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CTERMEQ, CTERMNE — CTERMEQ</td>
</tr>
</tbody>
</table>
### SVE pointer conflict compare

These instructions are under [SVE Integer Compare - Scalars](#).

<table>
<thead>
<tr>
<th>Field</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>0 0 1 0 1 0 1 0 1</td>
<td>Size 1</td>
</tr>
</tbody>
</table>

### SVE Integer Wide Immediate - Unpredicated

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Field</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>00100101</td>
<td>1 op0 op1 11</td>
</tr>
</tbody>
</table>

### SVE integer add/subtract immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

<table>
<thead>
<tr>
<th>Field</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>0 0 1 0 0 1 0 1</td>
<td>Size 1 0 0</td>
</tr>
</tbody>
</table>

### SVE integer min/max immediate (unpredicated)

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

<table>
<thead>
<tr>
<th>Field</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>0 0 1 0 0 0 1 0 1</td>
<td>Size 1 0 1</td>
</tr>
</tbody>
</table>
**SVE integer multiply immediate (unpredicated)**

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

```
| 0xx | 1 | UNALLOCATED |
| 000 | 0 | SMAX (immediate) |
| 001 | 0 | UMAX (immediate) |
| 010 | 0 | SMIN (immediate) |
| 011 | 0 | UMIN (immediate) |
| 1xx | | UNALLOCATED |
```

**SVE broadcast integer immediate (unpredicated)**

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

```
| 000 | 0 | MUL (immediate) |
| 000 | 1 | UNALLOCATED |
| 001 | | UNALLOCATED |
| 01x | | UNALLOCATED |
| 1xx | | UNALLOCATED |
```

**SVE broadcast floating-point immediate (unpredicated)**

These instructions are under [SVE Integer Wide Immediate - Unpredicated](#).

```
| 00 | DUP (immediate) |
| 01 | UNALLOCATED |
| 1x | UNALLOCATED |
```

**SVE predicate count**

These instructions are under [SVE encodings](#).
Top-level encodings for A64

### SVE Inc/Dec by Predicate Count

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>o2</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td></td>
</tr>
<tr>
<td>01x</td>
<td></td>
</tr>
<tr>
<td>1xx</td>
<td></td>
</tr>
</tbody>
</table>

### SVE saturating inc/dec vector by predicate count

These instructions are under **SVE Inc/Dec by Predicate Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### SVE saturating inc/dec register by predicate count

These instructions are under **SVE Inc/Dec by Predicate Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>U</td>
</tr>
<tr>
<td>01</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### SVE saturating inc/dec scalar by predicate count

These instructions are under **SVE Inc/Dec by Predicate Count**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>U</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
### SVE inc/dec vector by predicate count

These instructions are under **SVE Inc/Dec by Predicate Count**.

### SVE inc/dec register by predicate count

These instructions are under **SVE Inc/Dec by Predicate Count**.

### SVE Write FFR

These instructions are under **SVE encodings**.
SVE FFR write from predicate

These instructions are under SVE Write FFR.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>WRFFR</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE FFR initialise

These instructions are under SVE Write FFR.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SETFFR</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Integer Multiply-Add - Unpredicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>SVE integer dot product (unpredicated)</td>
</tr>
<tr>
<td>0001x</td>
<td>SVE2 saturating multiply-add interleaved long</td>
</tr>
<tr>
<td>001xx</td>
<td>CDOT (vectors)</td>
</tr>
<tr>
<td>01xxx</td>
<td>SVE2 complex integer multiply-add</td>
</tr>
<tr>
<td>10xxx</td>
<td>SVE2 integer multiply-add long</td>
</tr>
<tr>
<td>110xx</td>
<td>SVE2 saturating multiply-add long</td>
</tr>
<tr>
<td>1110x</td>
<td>SVE2 saturating multiply-add high</td>
</tr>
<tr>
<td>11110</td>
<td>SVE mixed sign dot product</td>
</tr>
<tr>
<td>11111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE integer dot product (unpredicated)

These instructions are under SVE Integer Multiply-Add - Unpredicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>op0</td>
</tr>
<tr>
<td>Decode fields</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>U 0</td>
<td>SDOT (vectors)</td>
</tr>
<tr>
<td>U 1</td>
<td>UDOT (vectors)</td>
</tr>
</tbody>
</table>

**SVE2 saturating multiply-add interleaved long**

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 0</td>
<td>SQDMLALBT</td>
</tr>
<tr>
<td>S 1</td>
<td>SQDMLSLBT</td>
</tr>
</tbody>
</table>

**SVE2 complex integer multiply-add**

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op 0</td>
<td>CMLA (vectors)</td>
</tr>
<tr>
<td>op 1</td>
<td>SQRDCMLAH (vectors)</td>
</tr>
</tbody>
</table>

**SVE2 integer multiply-add long**

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S U T 0 0 0</td>
<td>SMLALB (vectors)</td>
</tr>
<tr>
<td>S U T 0 0 1</td>
<td>SMLALT (vectors)</td>
</tr>
<tr>
<td>S U T 0 1 0</td>
<td>UMLALB (vectors)</td>
</tr>
<tr>
<td>S U T 0 1 1</td>
<td>UMLALT (vectors)</td>
</tr>
<tr>
<td>S U T 1 0 0</td>
<td>SMLSLB (vectors)</td>
</tr>
<tr>
<td>S U T 1 0 1</td>
<td>SMLSLT (vectors)</td>
</tr>
<tr>
<td>S U T 1 1 0</td>
<td>UMLSLB (vectors)</td>
</tr>
<tr>
<td>S U T 1 1 1</td>
<td>UMLSLT (vectors)</td>
</tr>
</tbody>
</table>

**SVE2 saturating multiply-add long**

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```
### SVE2 saturating multiply-add high

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 0 T 0</td>
<td>SQDMLALB (vectors)</td>
</tr>
<tr>
<td>S 0 T 1</td>
<td>SQDMLALT (vectors)</td>
</tr>
<tr>
<td>S 1 T 0</td>
<td>SQDMLSLB (vectors)</td>
</tr>
<tr>
<td>S 1 T 1</td>
<td>SQDMLSLT (vectors)</td>
</tr>
</tbody>
</table>

### SVE mixed sign dot product

These instructions are under [SVE Integer Multiply-Add - Unpredicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 0</td>
<td>SQRDMLAH (vectors)</td>
</tr>
<tr>
<td>S 1</td>
<td>SQRDMLSH (vectors)</td>
</tr>
</tbody>
</table>

### SVE2 Integer - Predicated

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size 0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>size 10</td>
<td>USDOT (vectors)</td>
</tr>
<tr>
<td>size 11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE2 integer pairwise add and accumulate long

These instructions are under [SVE Integer - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 0010</td>
<td>SVE2 integer pairwise add and accumulate long</td>
</tr>
<tr>
<td>op0 0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op0 011x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>op1 0x0x</td>
<td>SVE2 integer unary operations (predicated)</td>
</tr>
<tr>
<td>op1 0xxx</td>
<td>SVE2 saturating/rounding bitwise shift left (predicated)</td>
</tr>
<tr>
<td>op1 10xx</td>
<td>SVE2 integer halving add/subtract (predicated)</td>
</tr>
<tr>
<td>op1 10x</td>
<td>SVE2 integer pairwise arithmetic</td>
</tr>
<tr>
<td>op1 11x</td>
<td>SVE2 saturating add/subtract</td>
</tr>
<tr>
<td>op1 11xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### SVE2 integer unary operations (predicated)

These instructions are under **SVE2 Integer - Predicated**.

### SVE2 saturating/rounding bitwise shift left (predicated)

These instructions are under **SVE2 Integer - Predicated**.

### SVE2 integer halving add/subtract (predicated)

These instructions are under **SVE2 Integer - Predicated**.
### SVE2 Integer pairwise arithmetic

These instructions are under [SVE2 Integer - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>R S U</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>SHADD</td>
</tr>
<tr>
<td>0 0 1</td>
<td>UHADD</td>
</tr>
<tr>
<td>0 1 0</td>
<td>SHSUB</td>
</tr>
<tr>
<td>0 1 1</td>
<td>UHSUB</td>
</tr>
<tr>
<td>1 0 0</td>
<td>SRHADD</td>
</tr>
<tr>
<td>1 0 1</td>
<td>URHADD</td>
</tr>
<tr>
<td>1 1 0</td>
<td>SHSUBR</td>
</tr>
<tr>
<td>1 1 1</td>
<td>UHSUBR</td>
</tr>
</tbody>
</table>

### SVE2 saturating add/subtract

These instructions are under [SVE2 Integer - Predicated](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc U</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 1</td>
<td>ADDP</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 0</td>
<td>SMAXP</td>
</tr>
<tr>
<td>10 1</td>
<td>UMAXP</td>
</tr>
<tr>
<td>11 0</td>
<td>SMINP</td>
</tr>
<tr>
<td>11 1</td>
<td>UMINP</td>
</tr>
</tbody>
</table>

### SVE Multiply - Indexed

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op S U</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>SQADD (vectors, predicated)</td>
</tr>
<tr>
<td>0 0 1</td>
<td>UQADD (vectors, predicated)</td>
</tr>
<tr>
<td>0 1 0</td>
<td>SOSUB (vectors, predicated)</td>
</tr>
<tr>
<td>0 1 1</td>
<td>UQSUB (vectors, predicated)</td>
</tr>
<tr>
<td>1 0 0</td>
<td>SUQADD</td>
</tr>
<tr>
<td>1 0 1</td>
<td>USQADD</td>
</tr>
<tr>
<td>1 1 0</td>
<td>SQSUBR</td>
</tr>
<tr>
<td>1 1 1</td>
<td>UQSUBR</td>
</tr>
</tbody>
</table>
### SVE integer dot product (indexed)

These instructions are under **SVE Multiply - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000x</td>
<td>SVE integer dot product (indexed)</td>
</tr>
<tr>
<td>00001x</td>
<td>SVE2 integer multiply-add (indexed)</td>
</tr>
<tr>
<td>00010x</td>
<td>SVE2 saturating multiply-add high (indexed)</td>
</tr>
<tr>
<td>00011x</td>
<td>SVE mixed sign dot product (indexed)</td>
</tr>
<tr>
<td>001xxx</td>
<td>SVE2 saturating multiply-add (indexed)</td>
</tr>
<tr>
<td>0100xx</td>
<td>SVE2 complex integer dot product (indexed)</td>
</tr>
<tr>
<td>0101xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0110xx</td>
<td>SVE2 complex integer multiply-add (indexed)</td>
</tr>
<tr>
<td>0111xx</td>
<td>SVE2 complex saturating multiply-add (indexed)</td>
</tr>
<tr>
<td>10xxx</td>
<td>SVE2 integer multiply-add long (indexed)</td>
</tr>
<tr>
<td>110xxx</td>
<td>SVE2 integer multiply long (indexed)</td>
</tr>
<tr>
<td>1110xx</td>
<td>SVE2 saturating multiply (indexed)</td>
</tr>
<tr>
<td>11110x</td>
<td>SVE2 saturating multiply high (indexed)</td>
</tr>
<tr>
<td>111110</td>
<td>SVE2 integer multiply (indexed)</td>
</tr>
<tr>
<td>111111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE2 integer multiply-add (indexed)

These instructions are under **SVE Multiply - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>0x</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

### SVE2 saturating multiply-add high (indexed)

These instructions are under **SVE Multiply - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>0x</td>
<td>0</td>
</tr>
<tr>
<td>0x</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>
### SVE mixed sign dot product (indexed)

These instructions are under **SVE Multiply - Indexed**.

```plaintext
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size S</td>
<td>SQRDMLAH (indexed)  — 16-bit</td>
</tr>
<tr>
<td>size S</td>
<td>SQRDMLSH (indexed)  — 16-bit</td>
</tr>
<tr>
<td>SQRDMLAH (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQRDMLSH (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQRDMLAH (indexed) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>SQRDMLSH (indexed) — 64-bit</td>
<td></td>
</tr>
</tbody>
</table>
```

### SVE2 saturating multiply-add (indexed)

These instructions are under **SVE Multiply - Indexed**.

```plaintext
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size S T</td>
<td>SQDMLALB (indexed)  — 32-bit</td>
</tr>
<tr>
<td>SQDMLALT (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLSLB (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLSLT (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLALB (indexed) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLALT (indexed) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLSLB (indexed) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLSLT (indexed) — 64-bit</td>
<td></td>
</tr>
</tbody>
</table>
```

### SVE2 complex integer dot product (indexed)

These instructions are under **SVE Multiply - Indexed**.

```plaintext
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size S rot T</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>SQDMLALB (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLALT (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLSLB (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLSLT (indexed) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLALB (indexed) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLALT (indexed) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLSLB (indexed) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>SQDMLSLT (indexed) — 64-bit</td>
<td></td>
</tr>
</tbody>
</table>
```
### CDOT (indexed)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>CDOT (indexed) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>CDOT (indexed) — 64-bit</td>
</tr>
</tbody>
</table>

**SVE2 complex integer multiply-add (indexed)**

These instructions are under [SVE Multiply - Indexed](#).

<table>
<thead>
<tr>
<th>Size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit</td>
<td>SVE2 complex saturating multiply-add (indexed)</td>
</tr>
<tr>
<td>32-bit</td>
<td>SVE2 integer multiply-add long (indexed)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>size</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### CMLA (indexed)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>CMLA (indexed) — 16-bit</td>
</tr>
<tr>
<td>11</td>
<td>CMLA (indexed) — 32-bit</td>
</tr>
</tbody>
</table>

### SQRDCMLAH (indexed)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>SQRDCMLAH (indexed) — 16-bit</td>
</tr>
<tr>
<td>11</td>
<td>SQRDCMLAH (indexed) — 32-bit</td>
</tr>
</tbody>
</table>

### SVE2 integer multiply-add long (indexed)

These instructions are under [SVE Multiply - Indexed](#).

<table>
<thead>
<tr>
<th>Size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit</td>
<td>SVE2 complex saturating multiply-add (indexed)</td>
</tr>
<tr>
<td>32-bit</td>
<td>SVE2 integer multiply-add long (indexed)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>SMLALB (indexed) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>SMLALT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>UMLALB (indexed) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>UMLALT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>SMLSLB (indexed) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>SMLSLT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>UMLSLB (indexed) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>UMLSLT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>SMLALB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>SMLALT (indexed) — 64-bit</td>
</tr>
<tr>
<td>10</td>
<td>UMLALB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>UMLALT (indexed) — 64-bit</td>
</tr>
<tr>
<td>10</td>
<td>SMLSLB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>SMLSLT (indexed) — 64-bit</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>Size</th>
<th>S</th>
<th>U</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Instruction Details

- **SMLSLT (indexed)** — 64-bit
- **UMLSLB (indexed)** — 64-bit
- **UMLSLT (indexed)** — 64-bit

#### SVE2 integer multiply long (indexed)

These instructions are under **SVE Multiply - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### SVE2 saturating multiply (indexed)

These instructions are under **SVE Multiply - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### SVE2 saturating multiply high (indexed)

These instructions are under **SVE Multiply - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
SVE2 integer multiply (indexed)

These instructions are under SVE Multiply - Indexed.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 0 0 size 1 opc 1 1 1 1 1 0 Zn Zd
```

SVE2 Widening Integer Arithmetic

These instructions are under SVE encodings.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
01000101 0 0 op0
```

SVE2 integer add/subtract long

These instructions are under SVE2 Widening Integer Arithmetic.

```
0 1 0 0 0 1 0 1 size 0 Zm 0 0 op S U T Zn Zd
```

SQRDMULH (indexed) — 64-bit

MUL (indexed) — 16-bit

MUL (indexed) — 32-bit

MUL (indexed) — 64-bit

SVE2 integer multiply long

SVE2 integer multiply wide

SVE2 integer multiply long

SADDLB

SADDLT

UADDLB

UADDLT

SSUBLB

SSUBLT

USUBLB

USUBLT
### SVE2 integer add/subtract wide

These instructions are under **SVE2 Widening Integer Arithmetic**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S  U  T</td>
<td></td>
</tr>
<tr>
<td>0  0  0</td>
<td>SADDWB</td>
</tr>
<tr>
<td>0  0  1</td>
<td>SADDWT</td>
</tr>
<tr>
<td>0  1  0</td>
<td>UADDWB</td>
</tr>
<tr>
<td>0  1  1</td>
<td>UADDWT</td>
</tr>
<tr>
<td>1  0  0</td>
<td>SSUBWB</td>
</tr>
<tr>
<td>1  0  1</td>
<td>SSUBWT</td>
</tr>
<tr>
<td>1  1  0</td>
<td>USUBWB</td>
</tr>
<tr>
<td>1  1  1</td>
<td>USUBWT</td>
</tr>
</tbody>
</table>

### SVE2 integer multiply long

These instructions are under **SVE2 Widening Integer Arithmetic**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op  U  T</td>
<td></td>
</tr>
<tr>
<td>0  0  0</td>
<td>SQDMULLB (vectors)</td>
</tr>
<tr>
<td>0  0  1</td>
<td>SQDMULLT (vectors)</td>
</tr>
<tr>
<td>0  1  0</td>
<td>PMULLB</td>
</tr>
<tr>
<td>0  1  1</td>
<td>PMULLT</td>
</tr>
<tr>
<td>1  0  0</td>
<td>SMULLB (vectors)</td>
</tr>
<tr>
<td>1  0  1</td>
<td>SMULLT (vectors)</td>
</tr>
<tr>
<td>1  1  0</td>
<td>UMULLB (vectors)</td>
</tr>
<tr>
<td>1  1  1</td>
<td>UMULLT (vectors)</td>
</tr>
</tbody>
</table>

### SVE Misc

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0 op1</td>
<td></td>
</tr>
<tr>
<td>01000101</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>op0 10xx SVE2 bitwise shift left long</td>
</tr>
<tr>
<td>0</td>
<td>op110xx UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00xx SVE2 integer add/subtract interleaved long</td>
</tr>
<tr>
<td>0</td>
<td>010xx SVE2 bitwise exclusive-or interleaved</td>
</tr>
<tr>
<td>0</td>
<td>0110 SVE integer matrix multiply accumulate</td>
</tr>
<tr>
<td>0</td>
<td>0111 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>11xx SVE2 bitwise permute</td>
</tr>
</tbody>
</table>
SVE2 bitwise shift left long

These instructions are under SVE Misc.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 0 0</td>
<td>SSHLLB</td>
</tr>
<tr>
<td>0 0</td>
<td>SSHLLT</td>
</tr>
<tr>
<td>1 0</td>
<td>USHLLB</td>
</tr>
<tr>
<td>1 1</td>
<td>USHLLT</td>
</tr>
</tbody>
</table>

SVE2 integer add/subtract interleaved long

These instructions are under SVE Misc.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 0 0</td>
<td>SADDLBT</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0</td>
<td>SSUBLBT</td>
</tr>
<tr>
<td>1 1</td>
<td>SSUBLTB</td>
</tr>
</tbody>
</table>

SVE2 bitwise exclusive-or interleaved

These instructions are under SVE Misc.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>tb 0</td>
<td>EORBT</td>
</tr>
<tr>
<td>1</td>
<td>EORTB</td>
</tr>
</tbody>
</table>

SVE integer matrix multiply accumulate

These instructions are under SVE Misc.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uns 00</td>
<td>SMMLA</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>USMMLA</td>
</tr>
<tr>
<td>11</td>
<td>UMMLA</td>
</tr>
</tbody>
</table>

SVE2 bitwise permute

These instructions are under SVE Misc.
### SVE2 Accumulate

These instructions are under SVE encodings.

#### Instruction Details

<table>
<thead>
<tr>
<th>OPC</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>BEXT</td>
</tr>
<tr>
<td>01</td>
<td>BDEP</td>
</tr>
<tr>
<td>10</td>
<td>BGRP</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>0000</td>
<td>011</td>
</tr>
<tr>
<td>!= 0000</td>
<td>011</td>
</tr>
<tr>
<td>00x</td>
<td>SVE2 integer add/subtract long with carry</td>
</tr>
<tr>
<td>010</td>
<td>SVE2 bitwise shift right and accumulate</td>
</tr>
<tr>
<td>10x</td>
<td>SVE2 bitwise shift and insert</td>
</tr>
<tr>
<td>111</td>
<td>SVE2 integer absolute difference and accumulate</td>
</tr>
</tbody>
</table>

#### SVE2 complex integer add

These instructions are under SVE2 Accumulate.

#### SVE2 integer absolute difference and accumulate long

These instructions are under SVE2 Accumulate.

#### SVE2 integer add/subtract long with carry

These instructions are under SVE2 Accumulate.
SVE2 bitwise shift right and accumulate

These instructions are under SVE2 Accumulate.

SVE2 bitwise shift and insert

These instructions are under SVE2 Accumulate.

SVE2 integer absolute difference and accumulate

These instructions are under SVE2 Accumulate.

SVE2 Narrowing

These instructions are under SVE encodings.
### SVE2 saturating extract narrow

These instructions are under *SVE2 Narrowing*.

![Decode fields](image)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SQXTNB</td>
</tr>
<tr>
<td>00 1</td>
<td>SQXTNT</td>
</tr>
<tr>
<td>01 0</td>
<td>UQXTNB</td>
</tr>
<tr>
<td>01 1</td>
<td>UQXTNT</td>
</tr>
<tr>
<td>10 0</td>
<td>SQXTUNB</td>
</tr>
<tr>
<td>10 1</td>
<td>SQXTUNT</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE2 bitwise shift right narrow

These instructions are under *SVE2 Narrowing*.

![Decode fields](image)

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0</td>
<td>SQSHRUNB</td>
</tr>
<tr>
<td>0 0 0 0 0 1</td>
<td>SQSHRUNT</td>
</tr>
<tr>
<td>0 0 0 1 0 0</td>
<td>SQRSHRUNB</td>
</tr>
<tr>
<td>0 0 0 1 0 1</td>
<td>SQRSHRUNT</td>
</tr>
<tr>
<td>0 1 0 0 0 0</td>
<td>SHRNB</td>
</tr>
<tr>
<td>0 1 0 0 1 0</td>
<td>SHRNT</td>
</tr>
<tr>
<td>0 1 1 0 0 0</td>
<td>BSHRNB</td>
</tr>
<tr>
<td>0 1 1 1 0</td>
<td>BSHRNT</td>
</tr>
<tr>
<td>1 0 0 0 0 0</td>
<td>SQSHRNB</td>
</tr>
<tr>
<td>1 0 0 0 0 1</td>
<td>SQSHRNT</td>
</tr>
<tr>
<td>1 0 1 0 0 0</td>
<td>SQRSHRNB</td>
</tr>
<tr>
<td>1 0 1 0 1 0</td>
<td>SQRSHRNT</td>
</tr>
<tr>
<td>1 1 0 0 0 0</td>
<td>UQSHRNB</td>
</tr>
<tr>
<td>1 1 0 0 0 1</td>
<td>UQSHRNT</td>
</tr>
<tr>
<td>1 1 1 0 0</td>
<td>UQRSHRNB</td>
</tr>
<tr>
<td>1 1 1 1 0</td>
<td>UQRSHRNT</td>
</tr>
</tbody>
</table>
SVE2 integer add/subtract narrow high part

These instructions are under SVE2 Narrowing.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>S  R  T</td>
<td></td>
</tr>
<tr>
<td>0  0  0</td>
<td>ADDHNB</td>
</tr>
<tr>
<td>0  0  1</td>
<td>ADDHNT</td>
</tr>
<tr>
<td>0  1  0</td>
<td>RADDHNB</td>
</tr>
<tr>
<td>0  1  1</td>
<td>RADDHNT</td>
</tr>
<tr>
<td>1  0  0</td>
<td>SUBHNB</td>
</tr>
<tr>
<td>1  0  1</td>
<td>SUBHNT</td>
</tr>
<tr>
<td>1  1  0</td>
<td>RSUBHNB</td>
</tr>
<tr>
<td>1  1  1</td>
<td>RSUBHNT</td>
</tr>
</tbody>
</table>

SVE2 character match

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>MATCH</td>
</tr>
<tr>
<td>1</td>
<td>NMATCH</td>
</tr>
</tbody>
</table>

SVE2 Histogram Computation - Segment

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>HISTSEG</td>
</tr>
<tr>
<td>!= 000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE2 Crypto Extensions

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op1  op2  op3</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>00  00  00000</td>
<td>SVE2 crypto unary operations</td>
</tr>
<tr>
<td>000</td>
<td>00  00  != 00000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>000</td>
<td>00  x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>000</td>
<td>01  0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>000</td>
<td>01  11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE2 crypto unary operations

These instructions are under SVE2 Crypto Extensions.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE2 crypto destructive binary operations

These instructions are under SVE2 Crypto Extensions.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE2 crypto constructive binary operations

These instructions are under SVE2 Crypto Extensions.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE floating-point convert precision odd elements

These instructions are under SVE encodings.
SVE floating-point pairwise operations

These instructions are under SVE encodings.

SVE floating-point multiply-add (indexed)

These instructions are under SVE encodings.

SVE floating-point complex multiply-add (indexed)

These instructions are under SVE encodings.
### SVE floating-point multiply (indexed)

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>FCMLA (indexed) — half-precision</td>
</tr>
<tr>
<td>11</td>
<td>FCMLA (indexed) — single-precision</td>
</tr>
</tbody>
</table>

### SVE Floating Point Widening Multiply-Add - Indexed

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>FMUL (indexed) — half-precision</td>
</tr>
<tr>
<td>10</td>
<td>FMUL (indexed) — single-precision</td>
</tr>
<tr>
<td>11</td>
<td>FMUL (indexed) — double-precision</td>
</tr>
</tbody>
</table>

### SVE BF16 floating-point dot product (indexed)

These instructions are under **SVE Floating Point Widening Multiply-Add - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>SVE BF16 floating-point dot product (indexed)</td>
</tr>
<tr>
<td>0 != 00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>SVE floating-point multiply-add long (indexed)</td>
</tr>
</tbody>
</table>

### SVE floating-point multiply-add long (indexed)

These instructions are under **SVE Floating Point Widening Multiply-Add - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>BFDOT (indexed)</td>
</tr>
</tbody>
</table>

### SVE BFloat16 floating-point dot product (indexed)

These instructions are under **SVE Floating Point Widening Multiply-Add - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 0 1 0 0 0</td>
<td>SVE BFloat16 floating-point dot product (indexed)</td>
</tr>
<tr>
<td>0 1 0 0 0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1</td>
<td>SVE floating-point multiply-add long (indexed)</td>
</tr>
</tbody>
</table>

### SVE floating-point multiply-add long (indexed)

These instructions are under **SVE Floating Point Widening Multiply-Add - Indexed**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>FMLALB (indexed)</td>
</tr>
</tbody>
</table>
### SVE Floating Point Widening Multiply-Add

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
<td>FMLALT (indexed)</td>
</tr>
<tr>
<td>0 1 0</td>
<td>FMLSLB (indexed)</td>
</tr>
<tr>
<td>0 1 1</td>
<td>FMLSLT (indexed)</td>
</tr>
<tr>
<td>1 0 0</td>
<td>BFMLALB (indexed)</td>
</tr>
<tr>
<td>1 0 1</td>
<td>BFMLALT (indexed)</td>
</tr>
<tr>
<td>1 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE BFloat16 floating-point dot product

These instructions are under **SVE Floating Point Widening Multiply-Add**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>SVE BFloat16 floating-point dot product</td>
</tr>
<tr>
<td>0 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>SVE floating-point multiply-add long</td>
</tr>
</tbody>
</table>

### SVE floating-point multiply-add long

These instructions are under **SVE Floating Point Widening Multiply-Add**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>BFDOT (vectors)</td>
</tr>
</tbody>
</table>

### SVE floating-point multiply-add long

These instructions are under **SVE Floating Point Widening Multiply-Add**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>BFDOT (vectors)</td>
</tr>
</tbody>
</table>

### SVE floating-point multiply-add long

These instructions are under **SVE Floating Point Widening Multiply-Add**.
SVE floating point matrix multiply accumulate

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>BFMMMLA</td>
</tr>
<tr>
<td>10</td>
<td>FMMLA — 32-bit element</td>
</tr>
<tr>
<td>11</td>
<td>FMMLA — 64-bit element</td>
</tr>
</tbody>
</table>

SVE floating-point compare vectors

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FCM&lt;cc&gt; (vectors) — FCMGE</td>
</tr>
<tr>
<td>01</td>
<td>FCM&lt;cc&gt; (vectors) — FCMGT</td>
</tr>
<tr>
<td>01</td>
<td>FCM&lt;cc&gt; (vectors) — FCMEQ</td>
</tr>
<tr>
<td>00</td>
<td>FCM&lt;cc&gt; (vectors) — FCMNE</td>
</tr>
<tr>
<td>10</td>
<td>FCMMUO</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>FAC&lt;cc&gt; — FACGT</td>
</tr>
</tbody>
</table>

SVE floating-point arithmetic (unpredicated)

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADD (vectors, unpredicated)</td>
</tr>
<tr>
<td>001</td>
<td>FSUB (vectors, unpredicated)</td>
</tr>
<tr>
<td>010</td>
<td>FMUL (vectors, unpredicated)</td>
</tr>
<tr>
<td>011</td>
<td>FTSMUL</td>
</tr>
<tr>
<td>10x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>FRECPS</td>
</tr>
<tr>
<td>111</td>
<td>FRSQRTS</td>
</tr>
</tbody>
</table>

SVE Floating Point Arithmetic - Predicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100101</td>
<td>0 op0</td>
</tr>
<tr>
<td>100</td>
<td>op1</td>
</tr>
<tr>
<td>00</td>
<td>op2</td>
</tr>
<tr>
<td>op0</td>
<td>op1</td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>0x</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>000</td>
</tr>
<tr>
<td>10</td>
<td>! =</td>
</tr>
<tr>
<td>11</td>
<td>0000</td>
</tr>
<tr>
<td>11</td>
<td>! =</td>
</tr>
</tbody>
</table>

**SVE floating-point arithmetic (predicated)**

These instructions are under SVE Floating Point Arithmetic - Predicated.

---

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FADD (vectors, predicted)</td>
</tr>
<tr>
<td>0001</td>
<td>FSUB (vectors, predicated)</td>
</tr>
<tr>
<td>0010</td>
<td>FMUL (vectors, predicated)</td>
</tr>
<tr>
<td>0011</td>
<td>FSUBR (vectors)</td>
</tr>
<tr>
<td>0100</td>
<td>FMAX (vectors)</td>
</tr>
<tr>
<td>0101</td>
<td>FMIN (vectors)</td>
</tr>
<tr>
<td>0111</td>
<td>FMIN (vectors)</td>
</tr>
<tr>
<td>1000</td>
<td>FABD</td>
</tr>
<tr>
<td>1001</td>
<td>FSCALE</td>
</tr>
<tr>
<td>1010</td>
<td>FMULX</td>
</tr>
<tr>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1100</td>
<td>FDIVR</td>
</tr>
<tr>
<td>1101</td>
<td>FDIV</td>
</tr>
<tr>
<td>111x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE floating-point arithmetic with immediate (predicated)

These instructions are under SVE Floating Point Arithmetic - Predicated.

---

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADD (immediate)</td>
</tr>
<tr>
<td>001</td>
<td>FSUB (immediate)</td>
</tr>
<tr>
<td>010</td>
<td>FMUL (immediate)</td>
</tr>
<tr>
<td>011</td>
<td>FSUBR (immediate)</td>
</tr>
<tr>
<td>100</td>
<td>FMAXNM (immediate)</td>
</tr>
<tr>
<td>101</td>
<td>FMINNM (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>FMAX (immediate)</td>
</tr>
<tr>
<td>111</td>
<td>FMIN (immediate)</td>
</tr>
</tbody>
</table>
SVE Floating Point Unary Operations - Predicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>SVE floating-point round to integral value</td>
</tr>
<tr>
<td>010</td>
<td>SVE floating-point convert precision</td>
</tr>
<tr>
<td>011</td>
<td>SVE floating-point unary operations</td>
</tr>
<tr>
<td>10x</td>
<td>SVE integer convert to floating-point</td>
</tr>
<tr>
<td>11x</td>
<td>SVE floating-point convert to integer</td>
</tr>
</tbody>
</table>

SVE floating-point round to integral value

These instructions are under SVE Floating Point Unary Operations - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FRINT&lt;r&gt; — nearest with ties to even</td>
</tr>
<tr>
<td>001</td>
<td>FRINT&lt;r&gt; — toward plus infinity</td>
</tr>
<tr>
<td>010</td>
<td>FRINT&lt;r&gt; — toward minus infinity</td>
</tr>
<tr>
<td>011</td>
<td>FRINT&lt;r&gt; — toward zero</td>
</tr>
<tr>
<td>100</td>
<td>FRINT&lt;r&gt; — nearest with ties to away</td>
</tr>
<tr>
<td>101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>FRINT&lt;r&gt; — current mode signalling inexact</td>
</tr>
<tr>
<td>111</td>
<td>FRINT&lt;r&gt; — current mode</td>
</tr>
</tbody>
</table>

SVE floating-point convert precision

These instructions are under SVE Floating Point Unary Operations - Predicated.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0 11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 10</td>
<td>FCVTX</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 00</td>
<td>FCVT — single-precision to half-precision</td>
</tr>
<tr>
<td>10 01</td>
<td>FCVT — half-precision to single-precision</td>
</tr>
<tr>
<td>10 10</td>
<td>BFCVT</td>
</tr>
<tr>
<td>11 00</td>
<td>FCVT — double-precision to half-precision</td>
</tr>
<tr>
<td>11 01</td>
<td>FCVT — half-precision to double-precision</td>
</tr>
<tr>
<td>11 10</td>
<td>FCVT — double-precision to single-precision</td>
</tr>
<tr>
<td>11 11</td>
<td>FCVT — single-precision to double-precision</td>
</tr>
</tbody>
</table>
SVE floating-point unary operations

These instructions are under **SVE Floating Point Unary Operations - Predicated**.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>FRECPX</td>
</tr>
<tr>
<td>01</td>
<td>FSQRT</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE integer convert to floating-point

These instructions are under **SVE Floating Point Unary Operations - Predicated**.

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>SCVTF — 16-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>UCVT — 16-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>SCVTF — 32-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>UCVT — 32-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>SCVTF — 64-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>UCVT — 64-bit to half-precision</td>
</tr>
<tr>
<td>0x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>SCVTF — 32-bit to single-precision</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>UCVT — 32-bit to single-precision</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>SCVTF — 32-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>SCVTF — 32-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>SCVTF — 64-bit to single-precision</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>SCVTF — 64-bit to single-precision</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>SCVTF — 64-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>UCVT — 64-bit to double-precision</td>
</tr>
</tbody>
</table>

SVE floating-point convert to integer

These instructions are under **SVE Floating Point Unary Operations - Predicated**.

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td></td>
<td>FLOGB</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>FCVTZS — half-precision to 16-bit</td>
</tr>
</tbody>
</table>
### SVE floating-point recursive reduction

These instructions are under SVE encodings.

### SVE Floating Point Unary Operations - Unpredicated

These instructions are under SVE encodings.

### SVE floating-point reciprocal estimate (unpredicated)

These instructions are under SVE Floating Point Unary Operations - Unpredicated.
SVE Floating Point Compare - with Zero

These instructions are under SVE encodings.

SVE floating-point compare with zero

These instructions are under SVE Floating Point Compare - with Zero.

SVE floating-point serial reduction (predicated)

These instructions are under SVE encodings.
SVE Floating Point Multiply-Add

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE floating-point multiply-accumulate writing addend</td>
</tr>
<tr>
<td>1</td>
<td>SVE floating-point multiply-accumulate writing multiplicand</td>
</tr>
</tbody>
</table>

SVE floating-point multiply-accumulate writing addend

These instructions are under SVE Floating Point Multiply-Add.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc 00</td>
<td>FMLA (vectors)</td>
</tr>
<tr>
<td>opc 01</td>
<td>FMLS (vectors)</td>
</tr>
<tr>
<td>opc 10</td>
<td>FNMLA</td>
</tr>
<tr>
<td>opc 11</td>
<td>FNMLS</td>
</tr>
</tbody>
</table>

SVE floating-point multiply-accumulate writing multiplicand

These instructions are under SVE Floating Point Multiply-Add.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc 00</td>
<td>FMAD</td>
</tr>
<tr>
<td>opc 01</td>
<td>FMSB</td>
</tr>
<tr>
<td>opc 10</td>
<td>FNMAAD</td>
</tr>
<tr>
<td>opc 11</td>
<td>FNMSB</td>
</tr>
</tbody>
</table>

SVE Memory - 32-bit Gather and Unsized Contiguous

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc op0 op1 op2 op3</td>
<td></td>
</tr>
<tr>
<td>00 x1 0xx 0</td>
<td>SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)</td>
</tr>
<tr>
<td>00 x1 0xx 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 x1 0xx 0</td>
<td>SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)</td>
</tr>
<tr>
<td>10 x1 0xx 0</td>
<td>SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)</td>
</tr>
<tr>
<td>11 0x 000 0</td>
<td>LDR (predicate)</td>
</tr>
<tr>
<td>11 0x 000 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 0x 010</td>
<td>LDR (vector)</td>
</tr>
</tbody>
</table>
### SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

![Decode fields]

<table>
<thead>
<tr>
<th>prfop</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

<table>
<thead>
<tr>
<th>Prfop</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

These instructions are under [SVE Memory - 32-bit Gather and Unsized Contiguous](#).

<table>
<thead>
<tr>
<th>Prfop</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
</tbody>
</table>
SVE contiguous prefetch (scalar plus immediate)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus immediate)</td>
</tr>
</tbody>
</table>

SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

The following constraints also apply to this encoding: opc != 11 && opc != 11

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LD1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>LDFF1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>LD1B (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>LDF1B (scalar plus vector)</td>
</tr>
<tr>
<td>00</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>LDF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>LDF1W (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE2 32-bit gather non-temporal load (scalar plus 32-bit unscaled offsets)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LDNT1SB</td>
</tr>
<tr>
<td>00</td>
<td>LDNT1B (vector plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>LDNT1SH</td>
</tr>
<tr>
<td>01</td>
<td>LDNT1H (vector plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>LDNT1W (vector plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE contiguous prefetch (scalar plus scalar)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE 32-bit gather prefetch (vector plus immediate)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (vector plus immediate)</td>
</tr>
</tbody>
</table>

SVE 32-bit gather load (vector plus immediate)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LD1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>LDFF1SB (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LD1B (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>LDFF1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LD1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LD1H (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LDFF1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>LD1W (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>LDFF1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE load and broadcast element

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

<table>
<thead>
<tr>
<th>dtypeh</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
### SVE Memory - Contiguous Load

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>LD1RB — 8-bit element</td>
</tr>
<tr>
<td>00 01</td>
<td>LD1RB — 16-bit element</td>
</tr>
<tr>
<td>00 10</td>
<td>LD1RB — 32-bit element</td>
</tr>
<tr>
<td>00 11</td>
<td>LD1RB — 64-bit element</td>
</tr>
<tr>
<td>01 00</td>
<td>LD1RSW</td>
</tr>
<tr>
<td>01 01</td>
<td>LD1RH — 16-bit element</td>
</tr>
<tr>
<td>01 10</td>
<td>LD1RH — 32-bit element</td>
</tr>
<tr>
<td>01 11</td>
<td>LD1RH — 64-bit element</td>
</tr>
<tr>
<td>10 00</td>
<td>LD1RSH — 64-bit element</td>
</tr>
<tr>
<td>10 01</td>
<td>LD1RSH — 32-bit element</td>
</tr>
<tr>
<td>10 10</td>
<td>LD1RW — 32-bit element</td>
</tr>
<tr>
<td>10 11</td>
<td>LD1RW — 64-bit element</td>
</tr>
<tr>
<td>11 00</td>
<td>LD1RSB — 64-bit element</td>
</tr>
<tr>
<td>11 01</td>
<td>LD1RSB — 32-bit element</td>
</tr>
<tr>
<td>11 10</td>
<td>LD1RSB — 16-bit element</td>
</tr>
<tr>
<td>11 11</td>
<td>LD1RD</td>
</tr>
</tbody>
</table>

---

**SVE contiguous non-temporal load (scalar plus immediate)**

These instructions are under SVE Memory - Contiguous Load.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>SVE contiguous non-temporal load (scalar plus immediate)</td>
</tr>
<tr>
<td>00 110</td>
<td>SVE contiguous non-temporal load (scalar plus scalar)</td>
</tr>
<tr>
<td>!= 00 001</td>
<td>SVE load multiple structures (scalar plus immediate)</td>
</tr>
<tr>
<td>!= 00 100</td>
<td>SVE load and broadcast quadword (scalar plus immediate)</td>
</tr>
<tr>
<td>1 001</td>
<td>SVE contiguous load (scalar plus immediate)</td>
</tr>
<tr>
<td>1 101</td>
<td>SVE contiguous non-fault load (scalar plus immediate)</td>
</tr>
<tr>
<td>1 111</td>
<td>SVE contiguous load (scalar plus scalar)</td>
</tr>
<tr>
<td>000</td>
<td>SVE load and broadcast quadword (scalar plus scalar)</td>
</tr>
<tr>
<td>010</td>
<td>SVE contiguous load (scalar plus scalar)</td>
</tr>
<tr>
<td>011</td>
<td>SVE contiguous first-fault load (scalar plus scalar)</td>
</tr>
</tbody>
</table>

---

**SVE contiguous load (scalar plus scalar)**

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td>LDNT1B (scalar plus immediate)</td>
</tr>
</tbody>
</table>
**Decode fields** | **msz** | **Instruction Details**
---|---|---
| 01 | LDNT1H (scalar plus immediate) |
| 10 | LDNT1W (scalar plus immediate) |
| 11 | LDNT1D (scalar plus immediate) |

**SVE contiguous non-temporal load (scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0  | msz 0 0  | Rm 1 1 0  | Pg  | Rn  | Zt
```

**Decode fields** | **msz** | **Instruction Details**
---|---|---
| 00 | LDNT1B (scalar plus scalar) |
| 01 | LDNT1H (scalar plus scalar) |
| 10 | LDNT1W (scalar plus scalar) |
| 11 | LDNT1D (scalar plus scalar) |

**SVE load multiple structures (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0  | msz |= 00 0 | imm4 1 1 1 | Pg  | Rn  | Zt
```

The following constraints also apply to this encoding: opc != 00 & opc != 00

**Decode fields** | **msz** | **opc** | **Instruction Details**
---|---|---|---
| 00 01 | LD2B (scalar plus immediate) |
| 00 10 | LD3B (scalar plus immediate) |
| 00 11 | LD4B (scalar plus immediate) |
| 01 01 | LD2H (scalar plus immediate) |
| 01 10 | LD3H (scalar plus immediate) |
| 01 11 | LD4H (scalar plus immediate) |
| 10 01 | LD2W (scalar plus immediate) |
| 10 10 | LD3W (scalar plus immediate) |
| 10 11 | LD4W (scalar plus immediate) |
| 11 01 | LD2D (scalar plus immediate) |
| 11 10 | LD3D (scalar plus immediate) |
| 11 11 | LD4D (scalar plus immediate) |

**SVE load multiple structures (scalar plus scalar)**

These instructions are under [SVE Memory - Contiguous Load](#).

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 0 0 1 0  | msz |= 00 0 | Rm 1 1 0  | Pg  | Rn  | Zt
```

The following constraints also apply to this encoding: opc != 00 & opc != 00
### SVE load and broadcast quadword (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz opc</td>
<td></td>
</tr>
<tr>
<td>00 01</td>
<td>LD2B (scalar plus scalar)</td>
</tr>
<tr>
<td>00 10</td>
<td>LD3B (scalar plus scalar)</td>
</tr>
<tr>
<td>00 11</td>
<td>LD4B (scalar plus scalar)</td>
</tr>
<tr>
<td>01 01</td>
<td>LD2H (scalar plus scalar)</td>
</tr>
<tr>
<td>01 10</td>
<td>LD3H (scalar plus scalar)</td>
</tr>
<tr>
<td>01 11</td>
<td>LD4H (scalar plus scalar)</td>
</tr>
<tr>
<td>10 01</td>
<td>LD2W (scalar plus scalar)</td>
</tr>
<tr>
<td>10 10</td>
<td>LD3W (scalar plus scalar)</td>
</tr>
<tr>
<td>10 11</td>
<td>LD4W (scalar plus scalar)</td>
</tr>
<tr>
<td>11 01</td>
<td>LD2D (scalar plus scalar)</td>
</tr>
<tr>
<td>11 10</td>
<td>LD3D (scalar plus scalar)</td>
</tr>
<tr>
<td>11 11</td>
<td>LD4D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE contiguous load (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype imm4 Pg Rn Zt</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>LD1B (scalar plus immediate) — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LD1B (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LD1B (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LD1B (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LD1SW (scalar plus immediate)</td>
</tr>
<tr>
<td>0101</td>
<td>LD1H (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LD1H (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LD1H (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LD1SH (scalar plus immediate) — 64-bit element</td>
</tr>
</tbody>
</table>
### SVE contiguous non-fault load (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>LD1SH (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td>LD1W (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LD1W (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td>LD1SB (scalar plus immediate) — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LD1SB (scalar plus immediate) — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LD1SB (scalar plus immediate) — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td>LD1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

### SVE load and broadcast quadword (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LDNF1B — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LDNF1B — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LDNF1B — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LDNF1B — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LDNF1SW</td>
</tr>
<tr>
<td>0101</td>
<td>LDNF1H — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LDNF1H — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LDNF1H — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LDNF1SH — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td>LDNF1SH — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td>LDNF1W — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LDNF1W — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td>LDNF1SB — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LDNF1SB — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LDNF1SB — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td>LDNP1D</td>
</tr>
</tbody>
</table>

---

Page 2910
### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>ssz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>00</td>
<td>LD1ROD (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>LD1ROD (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE contiguous load (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

### Decode fields

<table>
<thead>
<tr>
<th>dtype</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LD1B (scalar plus scalar) — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LD1B (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LD1B (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LD1B (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LD1SW (scalar plus scalar)</td>
</tr>
<tr>
<td>0101</td>
<td>LD1H (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LD1H (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LD1H (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LD1SH (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td>LD1SH (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td>LD1W (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LD1W (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1100</td>
<td>LD1SB (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LD1SB (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LD1SB (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td>LD1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE contiguous first-fault load (scalar plus scalar)

These instructions are under [SVE Memory - Contiguous Load](#).

### Decode fields

<table>
<thead>
<tr>
<th>dtype</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LDFF1B (scalar plus scalar) — 8-bit element</td>
</tr>
<tr>
<td>0001</td>
<td>LDFF1B (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>0010</td>
<td>LDFF1B (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>0011</td>
<td>LDFF1B (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>0100</td>
<td>LDFF1SW (scalar plus scalar)</td>
</tr>
<tr>
<td>0101</td>
<td>LDFF1H (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>0110</td>
<td>LDFF1H (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>0111</td>
<td>LDFF1H (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1000</td>
<td>LDFF1SH (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1001</td>
<td>LDFF1SH (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1010</td>
<td>LDFF1W (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1011</td>
<td>LDFF1W (scalar plus scalar) — 64-bit element</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>dtye</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>LDFF1SB (scalar plus scalar) — 64-bit element</td>
</tr>
<tr>
<td>1101</td>
<td>LDFF1SB (scalar plus scalar) — 32-bit element</td>
</tr>
<tr>
<td>1110</td>
<td>LDFF1SB (scalar plus scalar) — 16-bit element</td>
</tr>
<tr>
<td>1111</td>
<td>LDFF1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE Memory - 64-bit Gather

These instructions are under [SVE encodings](#).

---

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>0xx</th>
<th>1</th>
<th>UNALLOCATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>11</td>
<td>1xx</td>
<td>0</td>
<td>SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>x1</td>
<td>0xx</td>
<td>0</td>
<td>SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)</td>
</tr>
<tr>
<td>!=</td>
<td>11</td>
<td>1xx</td>
<td>SVE 64-bit gather load (scalar plus 64-bit scaled offsets)</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>x1</td>
<td>0xx</td>
<td>SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>101</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>111</td>
<td>0</td>
<td>SVE 64-bit gather prefetch (vector plus immediate)</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>111</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1x0</td>
<td></td>
<td>SVE2 64-bit gather non-temporal load (scalar plus unpacked 32-bit unscaled offsets)</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1xx</td>
<td></td>
<td>SVE 64-bit gather load (vector plus immediate)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1xx</td>
<td></td>
<td>SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)</td>
<td></td>
</tr>
<tr>
<td>x0</td>
<td>0xx</td>
<td></td>
<td>SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)</td>
<td></td>
</tr>
</tbody>
</table>

### SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>00</th>
<th>00</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>Zm</th>
<th>1</th>
<th>msz</th>
<th>Pg</th>
<th>Rn</th>
<th>0</th>
<th>prfop</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>xs</th>
<th>1</th>
<th>Zm</th>
<th>0</th>
<th>msz</th>
<th>Pg</th>
<th>Rn</th>
<th>0</th>
<th>prfop</th>
</tr>
</thead>
</table>

---

Top-level encodings for A64
<table>
<thead>
<tr>
<th>Decoded fields msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>PRFB (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (scalar plus vector)</td>
</tr>
</tbody>
</table>

**SVE 64-bit gather load (scalar plus 64-bit scaled offsets)**

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>Decode fields opc, U, ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 0 0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11 1 1</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

**SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)**

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>Decode fields opc, U, ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 0 0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
</tbody>
</table>
### SVE 64-bit gather prefetch (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>U</td>
</tr>
</tbody>
</table>

### SVE2 64-bit gather non-temporal load (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>PRFH (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>PRFW (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>PRFD (vector plus immediate)</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td>U</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>msz</td>
<td>U</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>
## SVE 64-bit gather load (vector plus immediate)

These instructions are under **SVE Memory - 64-bit Gather**.

<table>
<thead>
<tr>
<th>Decoded fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 0 1</td>
<td>LDFF1SW (vector plus immediate)</td>
</tr>
<tr>
<td>10 1 0</td>
<td>LD1W (vector plus immediate)</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDFF1W (vector plus immediate)</td>
</tr>
<tr>
<td>11 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 0</td>
<td>LD1D (vector plus immediate)</td>
</tr>
<tr>
<td>11 1 1</td>
<td>LDFF1D (vector plus immediate)</td>
</tr>
</tbody>
</table>

## SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)

These instructions are under **SVE Memory - 64-bit Gather**.

<table>
<thead>
<tr>
<th>Decoded fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td>LD1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00 0 1</td>
<td>LDFF1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00 1 0</td>
<td>LD1B (scalar plus vector)</td>
</tr>
<tr>
<td>00 1 1</td>
<td>LDFF1B (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11 1 1</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

## SVE 64-bit gather load (vector plus unpacked 32-bit unscaled offsets)

These instructions are under **SVE Memory - 64-bit Gather**.

<table>
<thead>
<tr>
<th>Decoded fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td>LD1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00 0 1</td>
<td>LDFF1SB (scalar plus vector)</td>
</tr>
<tr>
<td>00 1 0</td>
<td>LD1B (scalar plus vector)</td>
</tr>
<tr>
<td>00 1 1</td>
<td>LDFF1B (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 0 1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>01 1 1</td>
<td>LDFF1H (scalar plus vector)</td>
</tr>
<tr>
<td>10 0 0</td>
<td>LD1SW (scalar plus vector)</td>
</tr>
</tbody>
</table>
### SVE Memory - Contiguous Store and Unsized Contiguous

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 0 1</td>
<td>LDFF1SW (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>10 1 1</td>
<td>LDFF1W (scalar plus vector)</td>
</tr>
<tr>
<td>11 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 0</td>
<td>LD1D (scalar plus vector)</td>
</tr>
<tr>
<td>11 1 1</td>
<td>LDFF1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE contiguous store (scalar plus scalar)

These instructions are under SVE Memory - Contiguous Store and Unsized Contiguous.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xx 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10x 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110 0 0</td>
<td>STR (predicate)</td>
</tr>
<tr>
<td>110 0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110 1</td>
<td>STR (vector)</td>
</tr>
<tr>
<td>!= 110 1</td>
<td>SVE contiguous store (scalar plus scalar)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 110 && opc != 110

### SVE Memory - Non-temporal and Multi-register Store

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>ST1B (scalar plus scalar)</td>
</tr>
<tr>
<td>01x</td>
<td>ST1H (scalar plus scalar)</td>
</tr>
<tr>
<td>10x</td>
<td>ST1W (scalar plus scalar)</td>
</tr>
<tr>
<td>11 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1</td>
<td>ST1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

### SVE Memory - Contiguous Store and Unsized Contiguous

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>ST1B (scalar plus scalar)</td>
</tr>
<tr>
<td>00 1</td>
<td>ST1H (scalar plus scalar)</td>
</tr>
<tr>
<td>01 0</td>
<td>ST1W (scalar plus scalar)</td>
</tr>
<tr>
<td>01 1</td>
<td>SVE contiguous store (vector plus scalar)</td>
</tr>
</tbody>
</table>

### SVE Memory - Non-temporal and Multi-register Store

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0 0</td>
<td>SVE2 64-bit scatter non-temporal store (vector plus scalar)</td>
</tr>
</tbody>
</table>
SVE contiguous non-temporal store (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (vector plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (vector plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (vector plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>STNT1D (vector plus scalar)</td>
</tr>
</tbody>
</table>

SVE contiguous non-temporal store (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (scalar plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>STNT1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

SVE2 32-bit scatter non-temporal store (vector plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (vector plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (vector plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (vector plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE store multiple structures (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (vector plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (vector plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (vector plus scalar)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00
### Decode fields msz opc | Instruction Details
---|---
00 01 | ST2B (scalar plus scalar)
00 10 | ST3B (scalar plus scalar)
00 11 | ST4B (scalar plus scalar)
01 01 | ST2H (scalar plus scalar)
01 10 | ST3H (scalar plus scalar)
01 11 | ST4H (scalar plus scalar)
10 01 | ST2W (scalar plus scalar)
10 10 | ST3W (scalar plus scalar)
10 11 | ST4W (scalar plus scalar)
11 01 | ST2D (scalar plus scalar)
11 10 | ST3D (scalar plus scalar)
11 11 | ST4D (scalar plus scalar)

### SVE Memory - Scatter with Optional Sign Extend

These instructions are under **SVE encodings**.

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)</td>
</tr>
<tr>
<td>10</td>
<td>SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)</td>
</tr>
<tr>
<td>11</td>
<td>SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)</td>
</tr>
</tbody>
</table>

### SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under **SVE Memory - Scatter with Optional Sign Extend**.

### Decode fields

<table>
<thead>
<tr>
<th>msz opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

These instructions are under **SVE Memory - Scatter with Optional Sign Extend**.

### Decode fields

<table>
<thead>
<tr>
<th>msz opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
</tbody>
</table>
SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE Memory - Scatter

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)</td>
</tr>
<tr>
<td>10</td>
<td>SVE 64-bit scatter store (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>SVE 32-bit scatter store (vector plus immediate)</td>
</tr>
</tbody>
</table>

SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)

These instructions are under SVE Memory - Scatter.

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>op0</td>
</tr>
<tr>
<td>00</td>
<td>SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)</td>
</tr>
<tr>
<td>01</td>
<td>SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)</td>
</tr>
<tr>
<td>10</td>
<td>SVE 64-bit scatter store (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>SVE 32-bit scatter store (vector plus immediate)</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus vector)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)

These instructions are under **SVE Memory - Scatter**.

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus vector)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus vector)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus vector)</td>
</tr>
</tbody>
</table>

### SVE 64-bit scatter store (vector plus immediate)

These instructions are under **SVE Memory - Scatter**.

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (vector plus immediate)</td>
</tr>
</tbody>
</table>

### SVE 32-bit scatter store (vector plus immediate)

These instructions are under **SVE Memory - Scatter**.

### Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE Memory - Contiguous Store with Immediate Offset

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>imm5</th>
<th>Pg</th>
<th>Zn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110010</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### SVE contiguous non-temporal store (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

![Instruction Details Table]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE contiguous non-temporal store (scalar plus immediate)</td>
</tr>
<tr>
<td>!= 00</td>
<td>SVE store multiple structures (scalar plus immediate)</td>
</tr>
<tr>
<td>0</td>
<td>SVE contiguous store (scalar plus immediate)</td>
</tr>
</tbody>
</table>

### SVE store multiple structures (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

![Instruction Details Table]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>STNT1B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>STNT1H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>STNT1W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>STNT1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00

### SVE contiguous store (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

![Instruction Details Table]

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 01</td>
<td>ST2B (scalar plus immediate)</td>
</tr>
<tr>
<td>00 10</td>
<td>ST3B (scalar plus immediate)</td>
</tr>
<tr>
<td>00 11</td>
<td>ST4B (scalar plus immediate)</td>
</tr>
<tr>
<td>01 01</td>
<td>ST2H (scalar plus immediate)</td>
</tr>
<tr>
<td>01 10</td>
<td>ST3H (scalar plus immediate)</td>
</tr>
<tr>
<td>01 11</td>
<td>ST4H (scalar plus immediate)</td>
</tr>
<tr>
<td>10 01</td>
<td>ST2W (scalar plus immediate)</td>
</tr>
<tr>
<td>10 10</td>
<td>ST3W (scalar plus immediate)</td>
</tr>
<tr>
<td>10 11</td>
<td>ST4W (scalar plus immediate)</td>
</tr>
<tr>
<td>11 01</td>
<td>ST2D (scalar plus immediate)</td>
</tr>
<tr>
<td>11 10</td>
<td>ST3D (scalar plus immediate)</td>
</tr>
<tr>
<td>11 11</td>
<td>ST4D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

---

[_decode_fields_table]: #/image.png
[decode_fields_table]: #/image.png
[opc]: opc
[imm4]: imm4
Decoding fields

<table>
<thead>
<tr>
<th>msz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>ST1B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>ST1H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>ST1W (scalar plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td>ST1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

### Data Processing -- Immediate

These instructions are under the **top-level**.

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>PC-rel. addressing</td>
</tr>
<tr>
<td>010</td>
<td>Add/subtract (immediate)</td>
</tr>
<tr>
<td>011</td>
<td>Add/subtract (immediate, with tags)</td>
</tr>
<tr>
<td>100</td>
<td>Logical (immediate)</td>
</tr>
<tr>
<td>101</td>
<td>Move wide (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>Bitfield</td>
</tr>
<tr>
<td>111</td>
<td>Extract</td>
</tr>
</tbody>
</table>

### PC-rel. addressing

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>op</th>
<th>immlo</th>
<th>immhi</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Add/subtract (immediate)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>immlo</th>
<th>sh</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Add/subtract (immediate)

These instructions are under **Data Processing -- Immediate**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ADD (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>ADDS (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SUB (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>SUBS (immediate) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>ADD (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>ADDS (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SUB (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>SUBS (immediate) — 64-bit</td>
</tr>
</tbody>
</table>
Add/subtract (immediate, with tags)

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>02</th>
<th>uimm6</th>
<th>op3</th>
<th>uimm4</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>02</th>
<th>uimm6</th>
<th>op3</th>
<th>uimm4</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ADDG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SUBG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Logical (immediate)

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td></td>
<td>AND (immediate) — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td></td>
<td>ORR (immediate) — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td></td>
<td>EOR (immediate) — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
<td>ANDS (immediate) — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1</td>
<td></td>
<td>AND (immediate) — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td></td>
<td>ORR (immediate) — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td>EOR (immediate) — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td></td>
<td>ANDS (immediate) — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Move wide (immediate)

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>hw</th>
<th>imm16</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0x</td>
<td>MOVN — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0x</td>
<td>MOVZ — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0x</td>
<td>MOVK — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>1x</td>
<td>MOVN — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0x</td>
<td>MOVZ — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0x</td>
<td>MOVK — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bitfield

These instructions are under Data Processing -- Immediate.
### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>N</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SBFM — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>BFM — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>UBFM — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>SBFM — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td>BFM — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>UBFM — 64-bit</td>
</tr>
</tbody>
</table>

### Extract

These instructions are under **Data Processing -- Immediate**.

### Branches, Exception Generating and System instructions

These instructions are under the [top-level](#)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>0xxx</td>
<td></td>
<td><strong>Conditional branch (immediate)</strong></td>
</tr>
<tr>
<td>110</td>
<td>0xxx</td>
<td></td>
<td><strong>System instructions with register argument</strong></td>
</tr>
<tr>
<td>110</td>
<td>01000000110010</td>
<td>11111</td>
<td><strong>Hints</strong></td>
</tr>
<tr>
<td>110</td>
<td>01000000110010</td>
<td>11111</td>
<td><strong>Barriers</strong></td>
</tr>
<tr>
<td>110</td>
<td>010000000xx0100</td>
<td></td>
<td><strong>PSTATE</strong></td>
</tr>
<tr>
<td>110</td>
<td>0100100xxxxxxx</td>
<td></td>
<td><strong>System with result</strong></td>
</tr>
<tr>
<td>110</td>
<td>0100x01xxxxxx</td>
<td></td>
<td><strong>System instructions</strong></td>
</tr>
<tr>
<td>110</td>
<td>0100x1xxxxxx</td>
<td></td>
<td><strong>System register move</strong></td>
</tr>
<tr>
<td>00</td>
<td>1xxxx</td>
<td></td>
<td><strong>Unconditional branch (register)</strong></td>
</tr>
<tr>
<td>00</td>
<td>0xxxx</td>
<td></td>
<td><strong>Unconditional branch (immediate)</strong></td>
</tr>
</tbody>
</table>

---

Page 2924
Conditional branch (immediate)

These instructions are under Branches, Exception Generating and System instructions.

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1 o0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>B.cond</td>
</tr>
<tr>
<td>0 1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
```

Exception generation

These instructions are under Branches, Exception Generating and System instructions.

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc op2 LL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>000 000 00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>000 000 01</td>
<td>SVC</td>
<td>-</td>
</tr>
<tr>
<td>000 000 10</td>
<td>HVC</td>
<td>-</td>
</tr>
<tr>
<td>000 000 11</td>
<td>SMC</td>
<td>-</td>
</tr>
<tr>
<td>001 000 x1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>001 000 00</td>
<td>BRK</td>
<td>-</td>
</tr>
<tr>
<td>001 000 1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010 000 x1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010 000 00</td>
<td>HLT</td>
<td>-</td>
</tr>
<tr>
<td>010 000 1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>011 000 00</td>
<td>TCANCEL</td>
<td>FEAT_TME</td>
</tr>
<tr>
<td>011 000 01</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>011 000 1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>100 000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>101 000 00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>101 000 01</td>
<td>DCPS1</td>
<td>-</td>
</tr>
<tr>
<td>101 000 10</td>
<td>DCPS2</td>
<td>-</td>
</tr>
<tr>
<td>101 000 11</td>
<td>DCPS3</td>
<td>-</td>
</tr>
<tr>
<td>111 000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
```

System instructions with register argument

These instructions are under Branches, Exception Generating and System instructions.
## Decode fields

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0000 000</td>
<td>WFET</td>
<td>FEAT_WFxT</td>
<td></td>
</tr>
<tr>
<td>0000 001</td>
<td>WFIT</td>
<td>FEAT_WFxT</td>
<td></td>
</tr>
<tr>
<td>0000 01x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0000 1xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

### Hints

These instructions are under [Branches, Exception Generating and System instructions](#).

### Barriers

These instructions are under [Branches, Exception Generating and System instructions](#).
## Decode fields

<table>
<thead>
<tr>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>! = 11111</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>11111</td>
<td></td>
<td>CLREX</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>11111</td>
<td></td>
<td>DMB</td>
<td>-</td>
</tr>
<tr>
<td>110</td>
<td>11111</td>
<td></td>
<td>ISB</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>! = 11111</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>11111</td>
<td></td>
<td>SB</td>
<td>-</td>
</tr>
<tr>
<td>xx0x</td>
<td>001</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>xx10</td>
<td>001</td>
<td>11111</td>
<td>DSB — Memory nXS barrier</td>
<td>FEAT XS</td>
</tr>
<tr>
<td>xx11</td>
<td>001</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>! = 0x00</td>
<td>100</td>
<td>11111</td>
<td>DSB — memory barrier</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>011</td>
<td>11111</td>
<td>TCOMMIT</td>
<td>FEAT TME</td>
</tr>
<tr>
<td>0000</td>
<td>100</td>
<td>11111</td>
<td>SSBB</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>001x</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01xx</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>100</td>
<td>11111</td>
<td>PSSBB</td>
<td>-</td>
</tr>
<tr>
<td>1xxx</td>
<td>011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### PSTATE

These instructions are under [Branches, Exception Generating and System instructions](#).

---

<table>
<thead>
<tr>
<th>op1</th>
<th>Decode fields</th>
<th>Rt</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>! = 11111</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>11111</td>
<td></td>
<td>MSR (immediate)</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>11111</td>
<td>CFINV</td>
<td>FEAT_FlagM</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>11111</td>
<td>XAFLAG</td>
<td>FEAT_FlagM2</td>
</tr>
<tr>
<td>000</td>
<td>010</td>
<td>11111</td>
<td>XAFLAG</td>
<td>FEAT_FlagM2</td>
</tr>
</tbody>
</table>

### System with result

These instructions are under [Branches, Exception Generating and System instructions](#).

---

<table>
<thead>
<tr>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 011</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>! = 0011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>001</td>
<td>! = 011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>0011</td>
<td>! = 000x011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>0011</td>
<td>0000</td>
<td>011</td>
<td>TSTART</td>
<td>FEAT TME</td>
</tr>
<tr>
<td>011</td>
<td>0011</td>
<td>0001</td>
<td>011</td>
<td>TTEST</td>
<td>FEAT TME</td>
</tr>
</tbody>
</table>
## System instructions

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>L</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SYS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SYSL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## System register move

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>L</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MSR (register)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MRS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Unconditional branch (register)

These instructions are under Branches, Exception Generating and System instructions.

<table>
<thead>
<tr>
<th>L</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>BR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key B, zero modifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>BLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>op2</td>
<td>op3</td>
<td>Rn</td>
<td>op4</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>---------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>000010</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, zero modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>000011</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000000</td>
<td>!=</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000000</td>
<td>0000</td>
<td>RET</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000010</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>RETAA, RETAB — RETAA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000011</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>RETAA, RETAB — RETAB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>11111</td>
<td>ERET</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000010</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>ERETTAA, ERETAB — ERETTAA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000011</td>
<td>!=</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Unconditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>Decode fields op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>11111</td>
<td>ERETTA, ERETAB — ERETAB</td>
<td>FEAT_PAuth</td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>!= 000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000 != 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000 != 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000 != 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000 != 11111</td>
<td>DRPS</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>011x</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>00000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>000010</td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key A, register modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>000011</td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key B, register modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>1xxxxxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>00000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>000010</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, register modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>000011</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, register modifier</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>11111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>101x</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11xx</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

### Compare and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>Decode fields op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>BL</td>
</tr>
</tbody>
</table>

Unconditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

![Unconditional branch (immediate)](#)

Compare and branch (immediate)
### Decode fields and Instruction Details

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>CBZ — 32-bit</td>
<td>CBNZ — 32-bit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>CBZ — 64-bit</td>
<td>CBNZ — 64-bit</td>
<td></td>
</tr>
</tbody>
</table>

### Test and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>b5</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>TBZ</td>
<td>TBNZ</td>
</tr>
</tbody>
</table>

### Loads and Stores

These instructions are under the [top-level](#).

<table>
<thead>
<tr>
<th>op0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>op1</td>
<td>op2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op0</th>
<th>01</th>
<th>0</th>
<th>00</th>
<th>000000</th>
<th>Advanced SIMD load/store multiple structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1</td>
<td>01</td>
<td>0xxxxx</td>
<td>Advanced SIMD load/store multiple structures (post-indexed)</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>0x</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>10</td>
<td>x00000</td>
<td>Advanced SIMD load/store single structure</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>11</td>
<td>Advanced SIMD load/store single structure (post-indexed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>x1xxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>xx1xxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>xxx1xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>x0</td>
<td>xxxxx1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>1x</td>
<td>1xxxxx</td>
<td>Load/store memory tags</td>
<td></td>
</tr>
<tr>
<td>1x00</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx00</td>
<td>0</td>
<td>0x</td>
<td>Load/store exclusive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx01</td>
<td>0x</td>
<td>Load register (literal)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx10</td>
<td>00</td>
<td>Load/store no-allocate pair (offset)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx10</td>
<td>01</td>
<td>Load/store register pair (post-indexed)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx10</td>
<td>10</td>
<td>Load/store register pair (offset)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx10</td>
<td>11</td>
<td>Load/store register pair (pre-indexed)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx11</td>
<td>0x</td>
<td>0xxxxx</td>
<td>Load/store register (unscaled immediate)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx11</td>
<td>0x</td>
<td>0xxxxx</td>
<td>Load/store register (immediate post-indexed)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx11</td>
<td>0x</td>
<td>0xxxxx</td>
<td>Load/store register (unprivileged)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xx11</td>
<td>0x</td>
<td>0xxxxx</td>
<td>Load/store register (immediate pre-indexed)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Advanced SIMD load/store multiple structures

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>opcode</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
</tr>
<tr>
<td>0</td>
<td>11xx</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
</tr>
<tr>
<td>1</td>
<td>11xx</td>
</tr>
</tbody>
</table>

Advanced SIMD load/store multiple structures (post-indexed)

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Rm</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
</tr>
</tbody>
</table>
### Advanced SIMD load/store single structure

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>L</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>Rm</td>
<td>1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>11xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>0000</td>
<td>ST4 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>0010</td>
<td>ST1 (multiple structures) — four registers, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>0100</td>
<td>ST3 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>0110</td>
<td>ST1 (multiple structures) — three registers, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>0111</td>
<td>ST1 (multiple structures) — one register, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>1000</td>
<td>ST2 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>1010</td>
<td>ST1 (multiple structures) — two registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0000</td>
<td>ST4 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>!!= 11111</td>
<td>0010</td>
<td>ST1 (multiple structures) — four registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0110</td>
<td>ST3 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0111</td>
<td>ST1 (multiple structures) — three registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0111</td>
<td>ST1 (multiple structures) — one register, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>!!= 11111</td>
<td>1000</td>
<td>ST2 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>!!= 11111</td>
<td>1010</td>
<td>ST1 (multiple structures) — two registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0000</td>
<td>LD4 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0010</td>
<td>LD1 (multiple structures) — four registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!!= 11111</td>
<td>0100</td>
<td>LD3 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0110</td>
<td>LD1 (multiple structures) — three registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>0111</td>
<td>LD1 (multiple structures) — one register, register offset</td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</td>
<td>1000</td>
<td>LD2 (multiple structures) — register offset</td>
</tr>
<tr>
<td>1</td>
<td>!!= 11111</td>
<td>1010</td>
<td>LD1 (multiple structures) — two registers, register offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0000</td>
<td>LD4 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0010</td>
<td>LD1 (multiple structures) — four registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0100</td>
<td>LD3 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0110</td>
<td>LD1 (multiple structures) — three registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0111</td>
<td>LD1 (multiple structures) — one register, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1000</td>
<td>LD2 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1010</td>
<td>LD1 (multiple structures) — two registers, immediate offset</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Decode fields</td>
<td>S</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---------------</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>01</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
</tr>
</tbody>
</table>
## Advanced SIMD load/store single structure (post-indexed)

These instructions are under **Loads and Stores**.

### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td></td>
<td>LD3R</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>000</td>
<td></td>
<td></td>
<td>LD2 (single structure) — 8-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td></td>
<td></td>
<td>LD4 (single structure) — 8-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>LD2 (single structure) — 16-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>LD4 (single structure) — 16-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>00</td>
<td></td>
<td>LD2 (single structure) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>LD2 (single structure) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>00</td>
<td></td>
<td>LD4 (single structure) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>10</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td></td>
<td></td>
<td>LD2R</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Instruction Details

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Decode fields</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>000</td>
<td></td>
<td></td>
<td>ST1 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>001</td>
<td></td>
<td></td>
<td>ST3 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST1 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST3 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST1 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>ST1 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>0</td>
<td>00</td>
<td>ST3 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>ST3 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Decode fields</td>
<td>opcode</td>
<td>S</td>
<td>size</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>000</td>
<td></td>
<td></td>
<td>ST1 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>001</td>
<td></td>
<td></td>
<td>ST3 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST1 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST3 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST1 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>ST1 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td></td>
<td>ST3 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>ST3 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>000</td>
<td></td>
<td></td>
<td>ST2 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>001</td>
<td></td>
<td></td>
<td>ST4 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST2 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST4 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST2 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>ST2 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td></td>
<td>ST4 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>ST4 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>000</td>
<td></td>
<td></td>
<td>ST2 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>001</td>
<td></td>
<td></td>
<td>ST4 (single structure) — 8-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>ST2 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>ST4 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>ST2 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>ST2 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td></td>
<td>ST4 (single structure) — 32-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>0</td>
<td>01</td>
<td>ST4 (single structure) — 64-bit, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>x1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>10</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>000</td>
<td></td>
<td></td>
<td>LD1 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>001</td>
<td></td>
<td></td>
<td>LD3 (single structure) — 8-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>010</td>
<td>x0</td>
<td></td>
<td>LD1 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>011</td>
<td>x0</td>
<td></td>
<td>LD3 (single structure) — 16-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td></td>
<td>LD1 (single structure) — 32-bit, register offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>100</td>
<td>0</td>
<td>01</td>
<td>LD1 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<td>Decode fields</td>
<td>opcode</td>
<td>S</td>
<td>size</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---------------</td>
<td>--------</td>
<td>---</td>
<td>------</td>
<td>---------------------</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td>LD3 (single structure) — 32-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>101</td>
<td>01</td>
<td>LD3 (single structure) — 64-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>!= 11111</td>
<td>110</td>
<td>0</td>
<td>LD1R — register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD3R — register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>000</td>
<td>0</td>
<td>LD1 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>001</td>
<td>0</td>
<td>LD3 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>LD1 (single structure) — 16-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>LD3 (single structure) — 16-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>LD1 (single structure) — 32-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td>LD1 (single structure) — 64-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>LD3 (single structure) — 32-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>LD3 (single structure) — 64-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>LD1R — immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD3R — immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>110</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>000</td>
<td>0</td>
<td>LD2 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>001</td>
<td>0</td>
<td>LD4 (single structure) — 8-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>010</td>
<td>x0</td>
<td>LD2 (single structure) — 16-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>011</td>
<td>x0</td>
<td>LD4 (single structure) — 16-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>00</td>
<td>LD2 (single structure) — 32-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>100</td>
<td>01</td>
<td>LD2 (single structure) — 64-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>00</td>
<td>LD4 (single structure) — 32-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>101</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit, register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>110</td>
<td>0</td>
<td>LD2R — register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>111</td>
<td>0</td>
<td>LD4R — register offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>000</td>
<td>0</td>
<td>LD2 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>001</td>
<td>0</td>
<td>LD4 (single structure) — 8-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>010</td>
<td>x0</td>
<td>LD2 (single structure) — 16-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>011</td>
<td>x0</td>
<td>LD4 (single structure) — 16-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>00</td>
<td>LD2 (single structure) — 32-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>01</td>
<td>LD2 (single structure) — 64-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>00</td>
<td>LD4 (single structure) — 32-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>101</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit, immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>110</td>
<td>0</td>
<td>LD2R — immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>111</td>
<td>0</td>
<td>LD4R — immediate offset</td>
<td></td>
</tr>
</tbody>
</table>

Load/store memory tags

These instructions are under [Loads and Stores](#).
### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>imm9</th>
<th>op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>STG — post-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>STG — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>STG — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>000000000</td>
<td>STZGM</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LDG</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>STZG — post-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>STZG — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>STZG — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>ST2G — post-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>ST2G — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>ST2G — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>STZ2G — post-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>STZ2G — signed offset</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>STZ2G — pre-index</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000000000</td>
<td>LDGM</td>
<td>FEAT_MTE</td>
<td></td>
</tr>
</tbody>
</table>

### Load/store exclusive

These instructions are under [Loads and Stores](#).

### Instruction Details

<table>
<thead>
<tr>
<th>size</th>
<th>Decode fields</th>
<th>Rs</th>
<th>o0</th>
<th>Rt2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>0 0 1 1 1111</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0 0 1 1 1111</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0 0 0 0 0</td>
<td>STXRB</td>
<td></td>
<td></td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0 0 0 0 1</td>
<td>STLXRB</td>
<td></td>
<td></td>
<td>FEAT_LSE</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0 0 1 1 1111</td>
<td>CASP, CASPA, CASPAL, CASPL</td>
<td>32-bit</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0 0 1 1 1111</td>
<td>CASP, CASPA, CASPAL, CASPL</td>
<td>32-bit</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0 1 0 0 0</td>
<td>LDXRB</td>
<td></td>
<td></td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0 1 0 1 1</td>
<td>LDAXRB</td>
<td></td>
<td></td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0 1 1 1 1111</td>
<td>CASP, CASPA, CASPAL, CASPL</td>
<td>32-bit</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0 1 1 1 1111</td>
<td>CASP, CASPA, CASPAL, CASPL</td>
<td>32-bit</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1 0 0 0 0</td>
<td>STLLRB</td>
<td></td>
<td></td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1 0 0 0 1</td>
<td>STLRB</td>
<td></td>
<td></td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1 0 1 0 1111</td>
<td>CASB, CASAB, CASALB, CASLB</td>
<td>CASB</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1 0 1 1 1111</td>
<td>CASB, CASAB, CASALB, CASLB</td>
<td>CASB</td>
<td>FEAT_LSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1 1 0 0 0</td>
<td>LDLARB</td>
<td></td>
<td></td>
<td>FEAT_LOR</td>
<td></td>
</tr>
</tbody>
</table>

---

Top-level encodings for A64
<table>
<thead>
<tr>
<th>size</th>
<th>o2</th>
<th>Decode fields</th>
<th>oL</th>
<th>o1</th>
<th>o0</th>
<th>Rt2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>LDARB</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASALB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STXRH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLXRH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLXRH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>STXRH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>STLXRH</td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STXRH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>CASH, CASAH, CASALH, CASLH — CASH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASH, CASAH, CASALH, CASLH — CASALH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>LDLARH</td>
<td>FEAT_LOR</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>LDLARH</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASH, CASAH, CASALH, CASLH — CASAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASH, CASAH, CASALH, CASLH — CASALH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CAS</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDLAR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>LDLAR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STXR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>STLXR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>STXR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDXR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LDAXR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>LDXR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>LDAXR</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASAL</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>o2</th>
<th>o1</th>
<th>o0</th>
<th>Rt2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>LDAXP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>STLLR — 64-bit</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>STLR — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CAS — 64-bit CAS</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CAS — 64-bit CASL</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>LDLAR — 64-bit</td>
<td>FEAT_LOR</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>LDAR — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CAS — 64-bit CASA</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CAS — 64-bit CASAL</td>
</tr>
</tbody>
</table>

### LDAPR/STLR (unscaled immediate)

These instructions are under **Loads and Stores**.

### Load register (literal)

These instructions are under **Loads and Stores**.
## Load/store no-allocate pair (offset)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>LDR (literal, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>PRFM (literal)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

## Load/store register pair (post-indexed)

These instructions are under [ Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>STP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>STGP</td>
<td>FEAT_MTE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDPSW</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>STP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDP (SIMD&amp;FP) — 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
Load/store register pair (offset)

These instructions are under Loads and Stores.

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load/store register pair (pre-indexed)

These instructions are under Loads and Stores.

<table>
<thead>
<tr>
<th>opc</th>
<th>V</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Load/store register (unscaled immediate)

These instructions are under Loads and Stores.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top-level encodings for A64
### Load/store register (immediate post-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDUR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDUR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STURH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRURH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRUSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRUSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDUR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STRUR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDUR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDURSW</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STRUR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDUR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STRUR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDUR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>PRFUM</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STRUR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDUR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
</tbody>
</table>
### Load/store register (unprivileged)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>

### Load/store register (immediate pre-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>STTRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDTRB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDTRSB — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDTRSB — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>STTRH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDTRH</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDTRSH — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDTRSH — 32-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STTR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDTR — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDTRSW</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STTR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDTR — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
## Atomic memory operations

These instructions are under [Loads and Stores](#).

### Top-level encodings for A64

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1 1 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 0 00</td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00 0 01</td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00 0 10</td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00 0 11</td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00 1 10</td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00 1 11</td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01 0 00</td>
<td>STRH (immediate)</td>
</tr>
<tr>
<td>01 0 01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01 0 10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01 0 11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x 0 11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x 1 1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 0 00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10 0 01</td>
<td>LDR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10 0 10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11 0 00</td>
<td>STR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11 0 01</td>
<td>LDR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11 0 10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 1 00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11 1 01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>

### Architecture Version

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o3</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>Rs</td>
<td>o3</td>
<td>opc</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADD</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>LDCLRAB, LDCLRAB, LDCLRAB, LDCLRAB — LDCLRAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDFORB, LDFORB, LDFORB, LDFORB — LDFORB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETAB, LDSETAB, LDSETLB — LDSETB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>LDMAXB, LDMAXB, LDMAXALB, LDMAXLB — LDMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDMINB, LDMINAB, LDMINAB, LDMINLB — LDMINB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01</td>
<td>LDUMAXB, LDUMAXB, LDUMAXLB, LDUMAXLB — LDUMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>LDUMINB, LDUMINAB, LDUMINAB, LDUMINLB — LDUMINB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRAB, LDCLRAB, LDCLRAB, LDCLRAB — LDCLRAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDFORB, LDFORB, LDFORB, LDFORB — LDFORB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETAB, LDSETAB, LDSETLB — LDSETB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>LDMAXB, LDMAXB, LDMAXALB, LDMAXLB — LDMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDMINB, LDMINAB, LDMINAB, LDMINLB — LDMINB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01</td>
<td>LDUMAXB, LDUMAXB, LDUMAXLB, LDUMAXLB — LDUMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>LDUMINB, LDUMINAB, LDUMINAB, LDUMINLB — LDUMINB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDLB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRAB, LDCLRAB, LDCLRAB, LDCLRAB — LDCLRAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDFORB, LDFORB, LDFORB, LDFORB — LDFORB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETAB, LDSETAB, LDSETLB — LDSETB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>LDMAXB, LDMAXB, LDMAXALB, LDMAXLB — LDMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDMINB, LDMINAB, LDMINAB, LDMINLB — LDMINB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>01</td>
<td>LDUMAXB, LDUMAXB, LDUMAXLB, LDUMAXLB — LDUMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>LDUMINB, LDUMINAB, LDUMINAB, LDUMINLB — LDUMINB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRAB, LDCLRAB, LDCLRAB, LDCLRAB — LDCLRAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDFORB, LDFORB, LDFORB, LDFORB — LDFORB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETAB, LDSETAB, LDSETLB — LDSETB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>LDMAXB, LDMAXB, LDMAXALB, LDMAXLB — LDMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDMINB, LDMINAB, LDMINAB, LDMINLB — LDMINB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>LDUMAXB, LDUMAXB, LDUMAXLB, LDUMAXLB — LDUMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>LDUMINB, LDUMINAB, LDUMINAB, LDUMINLB — LDUMINB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>Rs</td>
<td>o3</td>
<td>opc</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>---------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>LDAPRB</td>
<td>FEAT_LRCPC</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>110</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>111</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPAB</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDAH, LDADDALH, LDADDLH — LDADD</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>Rs</td>
<td>o3</td>
<td>opc</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINLH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXLH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINLH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPLH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>LDAPRH</td>
<td>LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPAH</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADD</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>Rs</td>
<td>o3</td>
<td>opc</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>----------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLR — 32-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>LDMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLR — 32-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLR — 32-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>Rs</td>
<td>o3</td>
<td>opc</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>0</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>0</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>0</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>0</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>LD64B</td>
<td>FEAT LS64 V</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>001</td>
<td>0</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>010</td>
<td>0</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>011</td>
<td>0</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWP</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>0</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>0</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>0</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>0</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>0</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>
### Load/store register (register offset)

These instructions are under [Loads and Stores](#).

#### Table: Load/store register (register offset)

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>o3</th>
<th>opc</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWPA</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
<td>100</td>
<td>LDAPR — 64-bit</td>
<td>FEAT_LRCP</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRAL — 64-bit LDCLRAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINAL</td>
<td>FEAT_LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWPA</td>
<td>FEAT_LSE</td>
</tr>
</tbody>
</table>

---

#### Table: Load/store register (register offset)

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>opc</th>
<th>option</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>! = 011</td>
<td>STRB (register) — extended register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>! = 011</td>
<td>STRB (register) — shifted register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>! = 011</td>
<td>LDRB (register) — extended register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>! = 011</td>
<td>LDRB (register) — shifted register</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>! = 011</td>
<td>LDRSB (register) — 64-bit with extended register offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>! = 011</td>
<td>LDRSB (register) — 64-bit with shifted register offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>! = 011</td>
<td>LDRSB (register) — 32-bit with extended register offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>! = 011</td>
<td>LDRSB (register) — 32-bit with shifted register offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>! = 011</td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>! = 011</td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>! = 011</td>
<td>LDR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>! = 011</td>
<td>LDR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td></td>
<td>STRH (register)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td></td>
<td>LDRH (register)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td></td>
<td>LDRSH (register) — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td></td>
<td>LDRSH (register) — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>Decode fields</td>
<td>opc</td>
<td>option</td>
<td>Instruction Details</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---------------</td>
<td>-----</td>
<td>--------</td>
<td>--------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1 00</td>
<td></td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1 01</td>
<td></td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>0 11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>1 1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0 00</td>
<td></td>
<td></td>
<td>STR (register) — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0 01</td>
<td></td>
<td></td>
<td>LDR (register) — 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0 10</td>
<td></td>
<td></td>
<td>LDRSW (register)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 00</td>
<td></td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 01</td>
<td></td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0 00</td>
<td></td>
<td></td>
<td>STR (register) — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0 01</td>
<td></td>
<td></td>
<td>LDR (register) — 64-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0 10</td>
<td></td>
<td></td>
<td>PRFM (register)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 00</td>
<td></td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 01</td>
<td></td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Load/store register (pac)**

These instructions are underLoads and Stores.

<table>
<thead>
<tr>
<th>size</th>
<th>Decode fields</th>
<th>opc</th>
<th>W</th>
<th>Rn</th>
<th>Rt</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STRH (immediate)</td>
</tr>
</tbody>
</table>

**Load/store register (unsigned immediate)**

These instructions are underLoads and Stores.

<table>
<thead>
<tr>
<th>size</th>
<th>Decode fields</th>
<th>opc</th>
<th>W</th>
<th>Rn</th>
<th>Rt</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>1 1x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDRB (immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDRSB (immediate) — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDRSB (immediate) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDR (immediate, SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LDR (immediate, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>01</td>
<td>0 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STRH (immediate)</td>
</tr>
</tbody>
</table>
Data Processing -- Register

These instructions are under the **top-level**.

Data-processing (2 source)

These instructions are under Data Processing -- Register.

---

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>LDRH (immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>LDRSH (immediate) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td>LDRSH (immediate) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 16-bit</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>LDRSW (immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td>STR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td>LDR (immediate) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td>PRFM (immediate)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td>STR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-bit</td>
</tr>
</tbody>
</table>

---

### Instruction Details

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0110</td>
<td></td>
<td>Data-processing (2 source)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0110</td>
<td></td>
<td>Data-processing (1 source)</td>
</tr>
<tr>
<td>0</td>
<td>0xxx</td>
<td></td>
<td></td>
<td>Logical (shifted register)</td>
</tr>
<tr>
<td>0</td>
<td>1xx0</td>
<td></td>
<td></td>
<td>Add/subtract (shifted register)</td>
</tr>
<tr>
<td>0</td>
<td>1xx1</td>
<td></td>
<td></td>
<td>Add/subtract (extended register)</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>000000</td>
<td></td>
<td>Add/subtract (with carry)</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>x00001</td>
<td></td>
<td>Rotate right into flags</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>xx0010</td>
<td></td>
<td>Evaluate into flags</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>xxxx0x</td>
<td></td>
<td>Conditional compare (register)</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>xxxx1x</td>
<td></td>
<td>Conditional compare (immediate)</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td></td>
<td></td>
<td>Conditional select</td>
</tr>
<tr>
<td>1</td>
<td>1xxx</td>
<td></td>
<td></td>
<td>Data-processing (3 source)</td>
</tr>
</tbody>
</table>

---

### Data-processing (3 source)
### Decode fields

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00011x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 001101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00111x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00001x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0001xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 001xxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 000000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 000010</td>
<td>UDIV — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 000011</td>
<td>SDIV — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 00010x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 001000</td>
<td>LSLV — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 001001</td>
<td>LSRV — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 001010</td>
<td>ASRV — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 001011</td>
<td>RORV — 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 001100</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 010x11</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 010000</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 010001</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 010010</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 010100</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 010101</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 010110</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 000000</td>
<td>SUBP</td>
<td></td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>1 0 000010</td>
<td>UDIV — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 000011</td>
<td>SDIV — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 000100</td>
<td>IRG</td>
<td></td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>1 0 000101</td>
<td>GMI</td>
<td></td>
<td>FEAT_MTE</td>
<td></td>
</tr>
<tr>
<td>1 0 001000</td>
<td>LSLV — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 001001</td>
<td>LSRV — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 001010</td>
<td>ASRV — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 001011</td>
<td>RORV — 64-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 001100</td>
<td>PACGA</td>
<td></td>
<td>FEAT_PAuth</td>
<td></td>
</tr>
<tr>
<td>1 0 010xx0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 010x0x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 010011</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 010111</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 000000</td>
<td>SUBPS</td>
<td></td>
<td>FEAT_MTE</td>
<td></td>
</tr>
</tbody>
</table>

---

### Data-processing (1 source)

These instructions are under Data Processing -- Register.
<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>Decode fields</th>
<th>opcode2</th>
<th>opcode</th>
<th>Rn</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1xxxxx</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>xxx1x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>xx1xx</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>x1xxx</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1xxxx</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00011x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 001xxx</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 01xxx</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00000</td>
<td>RBIT</td>
<td>32-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00001</td>
<td>REV16</td>
<td>32-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00010</td>
<td>REV</td>
<td>32-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00011</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00100</td>
<td>CLZ</td>
<td>32-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00101</td>
<td>CLS</td>
<td>32-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00000</td>
<td>RBIT</td>
<td>64-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00001</td>
<td>REV16</td>
<td>64-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00010</td>
<td>REV32</td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00011</td>
<td>REV</td>
<td>64-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00000 00100</td>
<td>CLZ</td>
<td>64-bit</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00000 00000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 00000</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 00010</td>
<td>PACDA, PACDZA — PACDA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 00011</td>
<td>PACDB, PACDZB — PACDB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 000100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 000101</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 000110</td>
<td>AUTDA, AUTDZA — AUTDA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 000111</td>
<td>AUTDB, AUTDZB — AUTDB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 001000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 001001</td>
<td>PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 001010</td>
<td>PACDA, PACDZA — PACDZA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 001011</td>
<td>PACDB, PACDZB — PACDZB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 001100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIZA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 001101</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIZB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 001110</td>
<td>AUTDA, AUTDZA — AUTDZA</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 001111</td>
<td>AUTDB, AUTDZB — AUTDZB</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 010000</td>
<td>XPACD, XPACI, XPACLRI — XPACI</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 010001</td>
<td>XPACD, XPACI, XPACLRI — XPACD</td>
<td>FEAT_PAuth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 01001x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00001 0101xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
Logical (shifted register)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>Decode fields</th>
<th>opcode2</th>
<th>opcode</th>
<th>Rn</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>00001</td>
<td>011xxx</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Add/subtract (shifted register)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Decode fields</th>
<th>shift</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>1xxxxx</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>ADD (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>BIC (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>ORR (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>ORN (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>EOR (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>EON (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>ANDS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>BICS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>ADD (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>BIC (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>ORR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>ORN (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>EOR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>EON (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>ANDS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>BICS (shifted register) — 64-bit</td>
</tr>
</tbody>
</table>
**Add/subtract (extended register)**

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>opt</th>
<th>1</th>
<th>Rm</th>
<th>option</th>
<th>imm3</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**Instruction Details**

<table>
<thead>
<tr>
<th>Description</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD (extended register) — 32-bit</td>
<td>0 0 0 0 00</td>
</tr>
<tr>
<td>ADDS (extended register) — 32-bit</td>
<td>0 0 1 0 00</td>
</tr>
<tr>
<td>SUB (extended register) — 32-bit</td>
<td>0 1 0 0 00</td>
</tr>
<tr>
<td>SUBS (extended register) — 32-bit</td>
<td>0 1 1 0 00</td>
</tr>
<tr>
<td>ADD (extended register) — 64-bit</td>
<td>1 0 0 0 00</td>
</tr>
<tr>
<td>ADDS (extended register) — 64-bit</td>
<td>1 0 1 0 00</td>
</tr>
<tr>
<td>SUB (extended register) — 64-bit</td>
<td>1 1 0 0 00</td>
</tr>
<tr>
<td>SUBS (extended register) — 64-bit</td>
<td>1 1 1 0 00</td>
</tr>
</tbody>
</table>

**Add/subtract (with carry)**

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

**Instruction Details**

<table>
<thead>
<tr>
<th>Description</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC — 32-bit</td>
<td>0 0 0</td>
</tr>
<tr>
<td>ADCS — 32-bit</td>
<td>0 0 1</td>
</tr>
<tr>
<td>SBC — 32-bit</td>
<td>0 1 0</td>
</tr>
<tr>
<td>SBCS — 32-bit</td>
<td>0 1 1</td>
</tr>
<tr>
<td>ADC — 64-bit</td>
<td>1 0 0</td>
</tr>
<tr>
<td>ADCS — 64-bit</td>
<td>1 0 1</td>
</tr>
<tr>
<td>SBC — 64-bit</td>
<td>1 1 0</td>
</tr>
<tr>
<td>SBCS — 64-bit</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

**Rotate right into flags**

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>imm6</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>o2</th>
<th>mask</th>
</tr>
</thead>
</table>

**Instruction Details**

<table>
<thead>
<tr>
<th>Description</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>RMIF</td>
<td>FEAT_FlagM</td>
</tr>
</tbody>
</table>

**Architecture Version**

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>RMIF</td>
<td>FEAT_FlagM</td>
</tr>
<tr>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Page 2957
Evaluate into flags

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!= 000000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>0 != 1101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>0 1 1101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>1 0 1101</td>
<td>SETF8, SETF16 — SETF8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000000</td>
<td>0 0 1101</td>
<td>SETF8, SETF16 — SETF16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

Conditional compare (register)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 0 0</td>
<td>CCMN (register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 0 0</td>
<td>CCMP (register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0 0 0</td>
<td>CCMN (register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0 0 0</td>
<td>CCMP (register) — 64-bit</td>
</tr>
</tbody>
</table>

Conditional compare (immediate)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>1 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 0 0</td>
<td>CCMN (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0 0 0</td>
<td>CCMP (immediate) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0 0 0</td>
<td>CCMP (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0 0 0</td>
<td>CCMP (immediate) — 64-bit</td>
</tr>
</tbody>
</table>
### Conditional select

These instructions are under **Data Processing -- Register**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>Rm</th>
<th>cond</th>
<th>op2</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>
| 1x | UNALLOCATED
| 1  | UNALLOCATED
| 0  | 0  | 0  | 0  | 0  | CSEL | 32-bit |
| 0  | 0  | 0  | 0  | 1  | CSINC | 32-bit |
| 0  | 1  | 0  | 0  | 0  | CSINV | 32-bit |
| 0  | 1  | 0  | 0  | 1  | CSNEG | 32-bit |
| 1  | 0  | 0  | 0  | 0  | CSEL | 64-bit |
| 1  | 0  | 0  | 0  | 1  | CSINC | 64-bit |
| 1  | 1  | 0  | 0  | 0  | CSINV | 64-bit |
| 1  | 1  | 0  | 0  | 1  | CSNEG | 64-bit |

### Data-processing (3 source)

These instructions are under **Data Processing -- Register**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op54</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>op31</th>
<th>Rm</th>
<th>o0</th>
<th>Ra</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>
| 00 | 010  | 1 | UNALLOCATED
| 00 | 011  | UNALLOCATED
| 00 | 100  | UNALLOCATED
| 00 | 110  | 1  | UNALLOCATED
| 00 | 111  | UNALLOCATED
| 01 | UNALLOCATED
| 1x | UNALLOCATED
| 0  | 00  | 000 | 0  | MADD | 32-bit |
| 0  | 00  | 000 | 1  | MSUB | 32-bit |
| 0  | 00  | 001 | 0  | UNALLOCATED
| 0  | 00  | 001 | 1  | UNALLOCATED
| 0  | 00  | 010 | 0  | UNALLOCATED
| 0  | 00  | 101 | 0  | UNALLOCATED
| 0  | 00  | 101 | 1  | UNALLOCATED
| 0  | 00  | 110 | 0  | UNALLOCATED
| 1  | 00  | 000 | 0  | MADD | 64-bit |
| 1  | 00  | 000 | 1  | MSUB | 64-bit |
| 1  | 00  | 001 | 0  | SMADDL
| 1  | 00  | 001 | 1  | SMSUBL
| 1  | 00  | 010 | 0  | SMULH
| 1  | 00  | 101 | 0  | UMAADDL
| 1  | 00  | 101 | 1  | UMSUBL
| 1  | 00  | 110 | 0  | UMULH

---

Page 2959
Data Processing -- Scalar Floating-Point and Advanced SIMD

These instructions are under the top-level.

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
<th>Architecture version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0100</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>Cryptographic AES</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxx0xxxx0</td>
<td>Cryptographic three-register SHA</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxx0xxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>Cryptographic two-register SHA</td>
<td>-</td>
</tr>
<tr>
<td>0110</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x0xx</td>
<td>xxxx0xxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>00</td>
<td>00xx</td>
<td>xxxx0xxxx1</td>
<td>Advanced SIMD scalar copy</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>01</td>
<td>00xx</td>
<td>xxxx0xxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>0111</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxxx0xxxx1</td>
<td>Advanced SIMD scalar three same FP16</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxxx0xxxx1</td>
<td>Advanced SIMD scalar three same extra</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>1111</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous FP16</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>xx0x</td>
<td>xxxx1xxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>xx0x</td>
<td>xxxx1xxxx1</td>
<td>Advanced SIMD scalar three same extra</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>xx0x</td>
<td>xxxx1xxxx1</td>
<td>Advanced SIMD scalar pairwise</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>xx1x</td>
<td>1xxxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>xx1x</td>
<td>x1xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>xx1x</td>
<td>xxxxxxx00</td>
<td>Advanced SIMD scalar three different</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>xx1x</td>
<td>xxxxxxx1</td>
<td>Advanced SIMD scalar three same</td>
<td>-</td>
</tr>
<tr>
<td>01x1</td>
<td>10</td>
<td>xxxxxxx1</td>
<td>Advanced SIMD scalar shift by immediate</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>11</td>
<td>xxxxxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>1x</td>
<td>xxxxxxx0</td>
<td>Advanced SIMD scalar x indexed element</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>0x</td>
<td>xx0xx</td>
<td>xxxx0xxxx0</td>
<td>Advanced SIMD table lookup</td>
<td>-</td>
</tr>
<tr>
<td>0x00</td>
<td>0x</td>
<td>xx0xx</td>
<td>xxxx0xxxx10</td>
<td>Advanced SIMD permute</td>
<td>-</td>
</tr>
<tr>
<td>0x10</td>
<td>0x</td>
<td>xx0xx</td>
<td>xxxx0xxxx0</td>
<td>Advanced SIMD extract</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>00</td>
<td>00xx</td>
<td>xxxx0xxxx1</td>
<td>Advanced SIMD copy</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>01</td>
<td>00xx</td>
<td>xxxx0xxxx1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>0111</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>10xx</td>
<td>xxxx0xxxx1</td>
<td>Advanced SIMD three same (FP16)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>10xx</td>
<td>xxxx01xxx1</td>
<td>Advanced SIMD two-register miscellaneous (FP16)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>xx0xx</td>
<td>xxxx1xxxx0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>xx0xx</td>
<td>xxxx1xxxx1</td>
<td>Advanced SIMD three-register extension</td>
<td>FEAT DotProd</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x100</td>
<td>00xxxxx10</td>
<td>Advanced SIMD two-register miscellaneous</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x110</td>
<td>00xxxxx10</td>
<td>Advanced SIMD across lanes</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0xx0</td>
<td>0x</td>
<td>x1xx</td>
<td>1xxxxxx10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
UNALLOCATED

Advanced SIMD three different

Advanced SIMD three same

Advanced SIMD modified immediate

Advanced SIMD shift by immediate

Advanced SIMD vector x indexed element

Advanced SIMD modified immediate

Conversion between floating-point and fixed-point

Conversion between floating-point and integer

Floating-point data-processing (1 source)

Floating-point data-processing (2 source)

Floating-point data-processing (3 source)

Floating-point data-processing (1 source)

Floating-point data-processing (2 source)

Floating-point data-processing (3 source)

Cryptographic three-register, imm2

Cryptographic three-register SHA 512

Cryptographic four-register

Cryptographic two-register SHA 512

Cryptographic three-register SHA 512

Floating-point immediate

Floating-point conditional compare

Floating-point conditional select

Top-level encodings for A64

Cryptographic AES

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

Cryptographic three-register SHA

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.
### Cryptographic two-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 000</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00 001</td>
<td>00</td>
<td>SHA1P</td>
<td></td>
</tr>
<tr>
<td>00 010</td>
<td>01</td>
<td>SHA1M</td>
<td></td>
</tr>
<tr>
<td>00 011</td>
<td>10</td>
<td>SHA1SU0</td>
<td></td>
</tr>
<tr>
<td>00 100</td>
<td>00</td>
<td>SHA256H</td>
<td></td>
</tr>
<tr>
<td>00 101</td>
<td>01</td>
<td>SHA256H2</td>
<td></td>
</tr>
<tr>
<td>00 110</td>
<td>10</td>
<td>SHA256SU1</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00000</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00 00001</td>
<td>00</td>
<td>SHA1U</td>
<td></td>
</tr>
<tr>
<td>00 00010</td>
<td>01</td>
<td>SHA256SU0</td>
<td></td>
</tr>
<tr>
<td>00 00011</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar three same FP16

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00000</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00 00001</td>
<td>00</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00 00010</td>
<td>01</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00 00011</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00 00100</td>
<td>10</td>
<td>DUP (element)</td>
<td></td>
</tr>
<tr>
<td>00 00101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00 00111</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

---

Top-level encodings for A64
### Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

#### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>a</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>FMULX</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>FCMEQ (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>111</td>
<td>FRECPS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>111</td>
<td>FRSORTS</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>FCMGE (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>FACGE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>FABD</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>FCMGT (register)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
<td>FACGT</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Data Processing -- Scalar Floating-Point and Advanced SIMD
### Decode fields opcode | Instruction Details | Architecture Version
---|---|---
1 0 11100 | FCVTAU (vector) | FEAT_FP16
1 0 11101 | UCVTF (vector, integer) | FEAT_FP16
1 1 01100 | FCMGE (zero) | FEAT_FP16
1 1 01101 | FCMLE (zero) | FEAT_FP16
1 1 01110 | UNALLOCATED | -
1 1 11010 | FCVTPU (vector) | FEAT_FP16
1 1 11011 | FCVTZU (vector, integer) | FEAT_FP16
1 1 11101 | FRSDRTE | FEAT_FP16
1 1 11111 | UNALLOCATED | -

#### Advanced SIMD scalar three same extra

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>001x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0000</td>
<td>SQRDMLAH (vector)</td>
<td>FEAT_RDM</td>
</tr>
<tr>
<td>1 0001</td>
<td>SQRDMLSH (vector)</td>
<td>FEAT_RDM</td>
</tr>
</tbody>
</table>

#### Advanced SIMD scalar two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0010x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1100x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x 011xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x 11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x 10110</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
## Advanced SIMD scalar pairwise

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00011</td>
<td>USQADD</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>SQABS</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>CMGT (zero)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>CMEQ (zero)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>CMLT (zero)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>ABS</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>SOXTN, SOXTN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>10110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>11101</td>
<td>SCVT (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01100</td>
<td>FCMGT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01101</td>
<td>FCMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>01110</td>
<td>FCMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>10100</td>
<td>FCVTPS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11101</td>
<td>FCVTZS (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11110</td>
<td>PRECFP</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11111</td>
<td>PRECFPX</td>
</tr>
<tr>
<td>1</td>
<td>00011</td>
<td>USQADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>SQNEG</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>CMGE (zero)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>CMLE (zero)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>NEG (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>SOXTN, SOXTN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UQXTN, UQXTN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>10110</td>
<td>FCVTXN, FCVTXN2</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td>FCVTNU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11011</td>
<td>FCVTMU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td>FCVTAU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td>UCVT (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01100</td>
<td>FCMGE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01101</td>
<td>FCMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td>FCVTPU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11011</td>
<td>FCVTZU (vector, integer)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td>FRSORTE</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar pairwise

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
### Advanced SIMD scalar three different

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

#### Decode fields

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x 01100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 11011</td>
<td>ADDP (scalar)</td>
</tr>
<tr>
<td>0 00 01100</td>
<td>FMAXNMP (scalar)</td>
</tr>
<tr>
<td>0 00 01101</td>
<td>FADDP (scalar)</td>
</tr>
<tr>
<td>0 00 01111</td>
<td>FMAXP (scalar)</td>
</tr>
<tr>
<td>0 10 01100</td>
<td>FMINNMP (scalar)</td>
</tr>
<tr>
<td>0 10 01111</td>
<td>FMINP (scalar)</td>
</tr>
<tr>
<td>0 11 01100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 11 01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 11011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0x 01100</td>
<td>FMAXNMP (scalar)</td>
</tr>
<tr>
<td>1 0x 01101</td>
<td>FADDP (scalar)</td>
</tr>
<tr>
<td>1 0x 01111</td>
<td>FMAXP (scalar)</td>
</tr>
<tr>
<td>1 1x 01100</td>
<td>FMINNMP (scalar)</td>
</tr>
<tr>
<td>1 1x 01111</td>
<td>FMINP (scalar)</td>
</tr>
<tr>
<td>1001</td>
<td>SQDMLAL, SQDMLAL2 (vector)</td>
</tr>
<tr>
<td>1011</td>
<td>SQDMLSL, SQDMLSL2 (vector)</td>
</tr>
<tr>
<td>1101</td>
<td>SQDMULL, SQDMULL2 (vector)</td>
</tr>
</tbody>
</table>

---

```csv
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1001</td>
<td>SQDMLAL, SQDMLAL2 (vector)</td>
</tr>
<tr>
<td>0 1011</td>
<td>SQDMLSL, SQDMLSL2 (vector)</td>
</tr>
<tr>
<td>0 1101</td>
<td>SQDMULL, SQDMULL2 (vector)</td>
</tr>
<tr>
<td>1 1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1101</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
```

---

Page 2966
## Advanced SIMD scalar three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>011xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>11011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00001</td>
<td>SQADD</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00101</td>
<td>SQSUB</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>CMGT (register)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00111</td>
<td>CMGE (register)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>SSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01001</td>
<td>SQRSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>QRSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01011</td>
<td>QSRSHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
<td>ADD (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10001</td>
<td>CMTST</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10110</td>
<td>SQDMULH (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11011</td>
<td>FMULX</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11100</td>
<td>FCMEQ (register)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11110</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x 11111</td>
<td>FRECPS</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11010</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11110</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x 11111</td>
<td>FRSORTS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UQADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00101</td>
<td>UQSUB</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>CMHI (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>CMHS (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>USHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>UQSHL (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>URSHL</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD scalar shift by immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!= 0000 00000</td>
<td>SSSHR</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 00010</td>
<td>SSRA</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 00100</td>
<td>SRSR</td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 00110</td>
<td>SRSRA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>!= 0000 00001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 11001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 11011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 11100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 11110</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Top-level encodings for A64

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01011</td>
<td>UQRSHL</td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>SUB (vector)</td>
</tr>
<tr>
<td>1</td>
<td>10001</td>
<td>CMEQ (register)</td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>10110</td>
<td>SQRDMULH (vector)</td>
</tr>
<tr>
<td>1</td>
<td>10111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x 11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x 11001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x 11010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x 11011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x 11100</td>
<td>FCMGE (register)</td>
</tr>
<tr>
<td>1</td>
<td>0x 11101</td>
<td>FACGE</td>
</tr>
<tr>
<td>1</td>
<td>0x 11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x 11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x 11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x 11001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x 11010</td>
<td>FABD</td>
</tr>
<tr>
<td>1</td>
<td>1x 11100</td>
<td>FCMGT (register)</td>
</tr>
<tr>
<td>1</td>
<td>1x 11101</td>
<td>FACGT</td>
</tr>
<tr>
<td>1</td>
<td>1x 11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x 11111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### Instruction Details

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>!= 0000 01000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 01010</td>
<td>SHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 01100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 01110</td>
<td>SQRSHL (immediate)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 10000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 10001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 10010</td>
<td>SQSHRN, SQSHRN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 10011</td>
<td>SQORSHRN, SQORSHRN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 11100</td>
<td>SCVT (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>!= 0000 11111</td>
<td>FCVTZS (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00000</td>
<td>USHR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00010</td>
<td>USRA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00100</td>
<td>URSHR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 00110</td>
<td>URSRA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01000</td>
<td>SRI</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01010</td>
<td>SLI</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01100</td>
<td>SQSHLU</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 01110</td>
<td>UQOSHL (immediate)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10000</td>
<td>SQSHRUN, SQSHRUN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10001</td>
<td>SQORSHRUN, SQORSHRUN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10010</td>
<td>UQOSHRN, UQOSHRN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 10011</td>
<td>UQRSHRN, UQRSHRN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 11100</td>
<td>UCVT (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 0000 11111</td>
<td>FCVTZU (vector, fixed-point)</td>
<td></td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar x indexed element

#### These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

### Top-level encodings for A64
### Advanced SIMD Table Lookup

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>op2</th>
<th>len</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>TBL — single register table</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>TBX — single register table</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>TBL — two register table</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>TBX — two register table</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>TBL — three register table</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>TBX — three register table</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>TBL — four register table</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>TBX — four register table</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD Permute

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
Advanced SIMD extract

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

```
<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

Advanced SIMD copy

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

```
<p>| Q | 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |</p>
<table>
<thead>
<tr>
<th>-------------------------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

Advanced SIMD three same (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

```
<p>| Q | 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |</p>
<table>
<thead>
<tr>
<th>-----------------------------------</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```
### Advanced SIMD two-register miscellaneous (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
### Advanced SIMD three-register extension

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

#### Instruction Details

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 11000</td>
<td>FRINTN (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 11001</td>
<td>FRINTM (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 11010</td>
<td>FCVTNS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 11011</td>
<td>FCVTMS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 11100</td>
<td>FCVTAS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 0 11101</td>
<td>SCVTF (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 01100</td>
<td>FCMGT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 01101</td>
<td>FCMEQ (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 01110</td>
<td>FCMLT (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 01111</td>
<td>FABS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11000</td>
<td>FRINTP (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11001</td>
<td>FRINTZ (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11010</td>
<td>FCVTPS (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11011</td>
<td>FCVTZS (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11100</td>
<td>FRECPE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0 1 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0 11000</td>
<td>FRINTA (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11001</td>
<td>FRINTX (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11010</td>
<td>FCVTNU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11011</td>
<td>FCVTMU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11100</td>
<td>FCVTAU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 0 11101</td>
<td>UCVTF (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 01100</td>
<td>FCMGE (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 01101</td>
<td>FCMLE (zero)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 01110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1 01111</td>
<td>FNEG (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1 11001</td>
<td>FRINTI (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11010</td>
<td>FCVTPU (vector)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11011</td>
<td>FCVTZU (vector, integer)</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11101</td>
<td>FRSQRTTE</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1 1 11111</td>
<td>FSQRT (vector)</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>

---

### Decode fields

<table>
<thead>
<tr>
<th>Q</th>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0 1 1 1 0</td>
<td>size</td>
<td>0</td>
<td>Rm</td>
</tr>
<tr>
<td>0x</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0010</td>
<td>SDOT (vector)</td>
<td>FEAT_DotProd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10 0011</td>
<td>USDOT (vector)</td>
<td>FEAT_I8MM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD two-register miscellaneous

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

#### Top-level encodings for A64

<table>
<thead>
<tr>
<th>Q</th>
<th>Decode fields</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10xx</td>
<td>FCMLA</td>
<td>FCMLA</td>
<td>FEAT.FCMA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11x0</td>
<td>FCADD</td>
<td>FCADD</td>
<td>FEAT.FCMA</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 1101</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00 1111</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01 1111</td>
<td>BFDOT (vector)</td>
<td>BFDOT (vector)</td>
<td>FEAT.BF16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10 0011</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10 1111</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11 1111</td>
<td>BFMLALB, BFMLAUT (vector)</td>
<td>BFMLALB, BFMLAUT (vector)</td>
<td>FEAT.BF16</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Q</th>
<th>Decode fields</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01 1101</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 01xx</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 011x</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 10 0100</td>
<td>SMMLA (vector)</td>
<td>SMMLA (vector)</td>
<td>FEAT.I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 10 0101</td>
<td>USMMLA (vector)</td>
<td>USMMLA (vector)</td>
<td>FEAT.I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 01 1101</td>
<td>BFMMMA</td>
<td>BFMMMA</td>
<td>FEAT.BF16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 10 0100</td>
<td>UMMMLA (vector)</td>
<td>UMMMLA (vector)</td>
<td>FEAT.I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 10 0101</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>U #</td>
<td>Decode fields</td>
<td>opcode</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>---------------</td>
<td>--------</td>
<td>---------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10010</td>
<td>XTN, XTN2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10011</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>SQXTN, SQXTN2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x10110</td>
<td>FCVTN, FCVTN2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x10111</td>
<td>FCVTL, FCVTL2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x11000</td>
<td>PRINTN (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x11001</td>
<td>PRINTM (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x11010</td>
<td>FCVTNS (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x11011</td>
<td>FCVTMS (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x11100</td>
<td>FCVTAS (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x11101</td>
<td>SCVT (vector, integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x11110</td>
<td>PRINT32Z (vector)</td>
<td>FEAT_FRINTTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x11111</td>
<td>PRINT64Z (vector)</td>
<td>FEAT_FRINTTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x01100</td>
<td>FCMGT (zero)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x01101</td>
<td>FCMEQ (zero)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x01110</td>
<td>FCMLT (zero)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x01111</td>
<td>FABS (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x11000</td>
<td>PRINTP (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x11001</td>
<td>PRINTZ (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x11010</td>
<td>FCVTPS (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x11011</td>
<td>FCVTZS (vector, integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x11100</td>
<td>URECPE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x11101</td>
<td>FRECPE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1x11111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10110</td>
<td>BFCVTN, BFCVTN2</td>
<td>FEAT_BF16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>REV32 (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00010</td>
<td>UADDP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00011</td>
<td>USQADD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00100</td>
<td>CLZ (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>UADALP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>SQNEG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>CMGE (zero)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>CMLE (zero)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>NEG (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>SQXTUN, SQXTUN2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td>SHLL, SHLL2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UQXTN, UQXTN2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x10110</td>
<td>FCVTXN, FCVTXN2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x10111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x11000</td>
<td>PRINTA (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x11001</td>
<td>PRINTX (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x11010</td>
<td>FCVTNU (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x11011</td>
<td>FCVTMU (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x11100</td>
<td>FCVTAU (vector)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x11101</td>
<td>UCVTF (vector, integer)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD across lanes

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x 11110</td>
<td>PRINT32X (vector)</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x 11111</td>
<td>PRINT64X (vector)</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00 00101</td>
<td>NOT</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01 00101</td>
<td>RBIT (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 00101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01100</td>
<td>FCMGE (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01101</td>
<td>FCMLE (zero)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 01111</td>
<td>FNEG (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11001</td>
<td>PRINTI (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11010</td>
<td>FCVTU (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11011</td>
<td>FCVTU (vector, integer)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11100</td>
<td>URSORTE</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11101</td>
<td>FRSORTE</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1x 11111</td>
<td>FSORT (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10 10110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

---

### Top-level encodings for A64

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>size</td>
<td>1 1 0 0 0</td>
<td>opcode</td>
<td>1 0</td>
</tr>
<tr>
<td>0</td>
<td>00011</td>
<td>SADDLV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>SMAXV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11010</td>
<td>SMINV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11011</td>
<td>ADDV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 01100</td>
<td>FMAXNMV — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00 01111</td>
<td>FMAXV — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 01100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01 01111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 01100</td>
<td>FMINNMV — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10 01111</td>
<td>FMINV — half-precision</td>
<td>FEAT_FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 01100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11 01111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>00011</td>
<td>UADDLV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>01010</td>
<td>UMAXV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>11010</td>
<td>UMINV</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>11011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01100</td>
<td>FMAXNMV — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01111</td>
<td>FMAXV — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01100</td>
<td>FMINNMV — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01111</td>
<td>FMINV — single-precision and double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Advanced SIMD three different

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SADDL, SADDL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SADDDW, SADDW2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SSUBL, SSUBL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SSUBW, SSUBW2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>ADDHN, ADDHN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SABAL, SABAL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SUBHN, SUBHN2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SABDL, SABDL2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>SMLAL, SMLAL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>SQDMLAL, SQDMLAL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>SMLSL, SMLSL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>SQDMLSL, SQDMLSL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td>SMULL, SMULL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>SODMULL, SODMULL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>PMULL, PMULL2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>UADDL, UADDL2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>UADDDW, UADDDW2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>USUBL, USUBL2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>USUBW, USUBW2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>RADDHN, RADDHN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>UABAL, UABAL2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>RSUBHN, RSUBHN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>UABDL, UABDL2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>UMLAL, UMLAL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>UMLSL, UMLSL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1100</td>
<td>UMULL, UMULL2 (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD three same

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000</td>
<td>SHADD</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>000001</td>
<td>SQADD</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>000010</td>
<td>SRHADD</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>001000</td>
<td>SHSUB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>001010</td>
<td>SQSUB</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>001100</td>
<td>CMGT (register)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>001111</td>
<td>CMGE (register)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>010000</td>
<td>SSHL</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>010001</td>
<td>SQSHL (register)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>010100</td>
<td>SRSRL</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>010110</td>
<td>QRSHL</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>011000</td>
<td>SMAX</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>011010</td>
<td>SMIN</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>011100</td>
<td>SABD</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>011111</td>
<td>SABA</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>100000</td>
<td>ADD (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>100010</td>
<td>CMTST</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>100100</td>
<td>MLA (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>100110</td>
<td>MUL (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>101000</td>
<td>SMAXP</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>101010</td>
<td>SMINP</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>101100</td>
<td>SQDMULH (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>101111</td>
<td>ADDP (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x 110000</td>
<td>FMAXNM (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x 110010</td>
<td>FMLA (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x 110100</td>
<td>PADD (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x 110110</td>
<td>FMULX</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x 111000</td>
<td>FCMEQ (register)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x 111100</td>
<td>FMAX (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0x 111111</td>
<td>FRECP</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00 000011</td>
<td>AND (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>00 111001</td>
<td>FMLAL, FMLAL2 (vector)</td>
<td>FEAT FHM</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01 000011</td>
<td>BIC (vector, register)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>01 11101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x 110000</td>
<td>FMINNML (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x 110010</td>
<td>FMLS (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x 110100</td>
<td>ESUB (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x 111010</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x 111101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x 111110</td>
<td>EMIN (vector)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1x 111111</td>
<td>FRSORTS</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>10 000011</td>
<td>ORR (vector, register)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Top-level encodings for A64
<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>11101</td>
<td>FMLSL, FMLSL2 (vector) — FMLSL</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>00011</td>
<td>ORN (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>11101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>UHADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UQADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00010</td>
<td>URHADD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00100</td>
<td>UHSUB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00101</td>
<td>UQSUB</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>CMHI (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00111</td>
<td>CMHS (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>USHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01001</td>
<td>UQSHL (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>URSHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>UQRSHL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01100</td>
<td>UMAX</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01101</td>
<td>UMIN</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01110</td>
<td>UABD</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01111</td>
<td>UABA</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>SUB (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10001</td>
<td>CMEQ (register)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>MLS (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td>PMUL</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UMAXP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10101</td>
<td>UMINP</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10110</td>
<td>SQRDMULH (vector)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11000</td>
<td>FMAXNMP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td>FDADDP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11011</td>
<td>FMUL (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td>FCMGE (register)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td>FACGE</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11110</td>
<td>FMAXP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11111</td>
<td>FDIV (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>00011</td>
<td>EOR (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>11001</td>
<td>FMLAL, FMLAL2 (vector) — FMLAL2</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>00011</td>
<td>BSL</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>11001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11000</td>
<td>FMNNMP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td>FABD</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11100</td>
<td>FCMGT (register)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td>FACGT</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11110</td>
<td>FMINP (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>00011</td>
<td>BIT</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11001</td>
<td>FMLSL, FMLSL2 (vector) — FMLSL2</td>
<td>FEAT_FHM</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>00011</td>
<td>BIF</td>
<td>-</td>
</tr>
</tbody>
</table>
## Advanced SIMD modified immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 1 11 11001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Advanced SIMD shift by immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>U 0 1 1 1 1 1 0</td>
<td>MOVI — 32-bit shifted immediate</td>
<td>-</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: immh != 0000 & immh != 0000

### Decode fields

<table>
<thead>
<tr>
<th>U 0 1 1 1 1 0</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01001</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1011x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0  00000</td>
<td>SSHOP</td>
</tr>
<tr>
<td>0  00010</td>
<td>SSRA</td>
</tr>
<tr>
<td>0  00100</td>
<td>SRSPHR</td>
</tr>
<tr>
<td>0  00110</td>
<td>SRRSA</td>
</tr>
<tr>
<td>0  01000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0  01010</td>
<td>SHL</td>
</tr>
<tr>
<td>0  01100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0  01110</td>
<td>SQRSH (immediate)</td>
</tr>
<tr>
<td>0  10000</td>
<td>SHR, SHRN2</td>
</tr>
<tr>
<td>0  10001</td>
<td>RSHRN, RSHRN2</td>
</tr>
<tr>
<td>0  10010</td>
<td>SQRSHRN, SQRSHRN2</td>
</tr>
<tr>
<td>0  10011</td>
<td>SQRSHRN, SQRSHRN2</td>
</tr>
<tr>
<td>0  10100</td>
<td>SSHL, SSHLL2</td>
</tr>
<tr>
<td>0  11100</td>
<td>SCVT (vector, fixed-point)</td>
</tr>
<tr>
<td>0  11111</td>
<td>FCVTS (vector, fixed-point)</td>
</tr>
<tr>
<td>1  00000</td>
<td>USHR</td>
</tr>
<tr>
<td>1  00010</td>
<td>USRA</td>
</tr>
<tr>
<td>1  00100</td>
<td>URSRH</td>
</tr>
<tr>
<td>1  00110</td>
<td>URSRA</td>
</tr>
<tr>
<td>1  01000</td>
<td>SRI</td>
</tr>
<tr>
<td>1  01010</td>
<td>SLI</td>
</tr>
<tr>
<td>1  01100</td>
<td>SOSHLU</td>
</tr>
<tr>
<td>1  01110</td>
<td>UOSHL (immediate)</td>
</tr>
<tr>
<td>1  10000</td>
<td>SQSHRUN, SQSHRUN2</td>
</tr>
<tr>
<td>1  10001</td>
<td>SQRSHRUN, SQRSHRUN2</td>
</tr>
<tr>
<td>1  10010</td>
<td>USHRN, UOSHRN2</td>
</tr>
<tr>
<td>1  10011</td>
<td>UQRSHRN, UQRSHRN2</td>
</tr>
<tr>
<td>1  10100</td>
<td>USBL, USBLL2</td>
</tr>
<tr>
<td>1  11100</td>
<td>UCVTF (vector, fixed-point)</td>
</tr>
<tr>
<td>1  11111</td>
<td>FCVTZU (vector, fixed-point)</td>
</tr>
</tbody>
</table>

### Advanced SIMD vector x indexed element

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.
<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0010</td>
<td></td>
<td>SMLAL, SMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td></td>
<td>SQDMLAL, SQDMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td></td>
<td>SMLSL, SMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td></td>
<td>SQDMLSL, SQDMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td></td>
<td>MUL (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td></td>
<td>SMULL, SMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td></td>
<td>SQDMLULL, SQDMLULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1100</td>
<td></td>
<td>SQDMLULH (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td></td>
<td>SQRDMULH (by element)</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td></td>
<td>SDOT (by element)</td>
<td>FEAT DotProd</td>
</tr>
<tr>
<td>0 0x</td>
<td>0000</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0x</td>
<td>0100</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 00</td>
<td>0001</td>
<td></td>
<td>FMLA (by element) — half-precision</td>
<td>FEAT FP16</td>
</tr>
<tr>
<td>0 00</td>
<td>0101</td>
<td></td>
<td>FMLS (by element) — half-precision</td>
<td>FEAT FP16</td>
</tr>
<tr>
<td>0 00</td>
<td>1001</td>
<td></td>
<td>FMUL (by element) — half-precision</td>
<td>FEAT FP16</td>
</tr>
<tr>
<td>0 00</td>
<td>1111</td>
<td></td>
<td>SUDOT (by element)</td>
<td>FEAT I8MM</td>
</tr>
<tr>
<td>0 01</td>
<td>0001</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 01</td>
<td>0101</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 01</td>
<td>1111</td>
<td></td>
<td>BFDOT (by element)</td>
<td>FEAT BF16</td>
</tr>
<tr>
<td>0 1x</td>
<td>0001</td>
<td></td>
<td>FMLA (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>0101</td>
<td></td>
<td>FMLS (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 1x</td>
<td>1001</td>
<td></td>
<td>FMUL (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 10</td>
<td>0000</td>
<td></td>
<td>FMLAL, FMLAL2 (by element) — FMLAL</td>
<td>FEAT FHM</td>
</tr>
<tr>
<td>0 10</td>
<td>0100</td>
<td></td>
<td>FMLSL, FMLSL2 (by element) — FMLSL</td>
<td>FEAT FHM</td>
</tr>
<tr>
<td>0 10</td>
<td>1111</td>
<td></td>
<td>USDOT (by element)</td>
<td>FEAT I8MM</td>
</tr>
<tr>
<td>0 11</td>
<td>0000</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 11</td>
<td>0100</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 11</td>
<td>1111</td>
<td></td>
<td>BFMLALB, BFMLALT (by element)</td>
<td>FEAT BF16</td>
</tr>
<tr>
<td>1 00</td>
<td>0000</td>
<td></td>
<td>MLA (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>0010</td>
<td></td>
<td>UMLAL, UMLAL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>0100</td>
<td></td>
<td>MLS (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>0110</td>
<td></td>
<td>UMLSL, UMLSL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>1010</td>
<td></td>
<td>UMULL, UMULL2 (by element)</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>1011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>1101</td>
<td></td>
<td>SQRDMLAH (by element)</td>
<td>FEAT RDM</td>
</tr>
<tr>
<td>1 00</td>
<td>1110</td>
<td></td>
<td>UDOT (by element)</td>
<td>FEAT DotProd</td>
</tr>
<tr>
<td>1 00</td>
<td>1111</td>
<td></td>
<td>SQRDMLSH (by element)</td>
<td>FEAT RDM</td>
</tr>
<tr>
<td>1 0x</td>
<td>1000</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0x</td>
<td>1100</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>0001</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>0011</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>0101</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>0111</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 00</td>
<td>1001</td>
<td></td>
<td>FMULX (by element) — half-precision</td>
<td>FEAT FP16</td>
</tr>
<tr>
<td>1 01</td>
<td>0xx1</td>
<td></td>
<td>FCMLA (by element)</td>
<td>FEAT FCMA</td>
</tr>
<tr>
<td>1 1x</td>
<td>1001</td>
<td></td>
<td>FMULX (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1 10</td>
<td>0xx1</td>
<td></td>
<td>FCMLA (by element)</td>
<td>FEAT FCMA</td>
</tr>
</tbody>
</table>
Cryptographic three-register, imm2

These instructions are under *Data Processing -- Scalar Floating-Point and Advanced SIMD*.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0001</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0101</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0111</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1100</td>
</tr>
</tbody>
</table>

**Cryptographic three-register SHA 512**

These instructions are under *Data Processing -- Scalar Floating-Point and Advanced SIMD*.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>SM3TT1A</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>01</td>
<td>SM3TT1B</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>10</td>
<td>SM3TT2A</td>
<td>FEAT SM3</td>
</tr>
<tr>
<td>11</td>
<td>SM3TT2B</td>
<td>FEAT SM3</td>
</tr>
</tbody>
</table>

**Cryptographic four-register**

These instructions are under *Data Processing -- Scalar Floating-Point and Advanced SIMD*.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op0</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>FOR3</td>
<td>FEAT SHA3</td>
</tr>
<tr>
<td>01</td>
<td>BCAX</td>
<td>FEAT SHA3</td>
</tr>
</tbody>
</table>
### Cryptographic two-register SHA 512

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

- **Op0**: SM3SS1
  - FEAT_SM3
- **Op0**: UNALLOCATED
  - -

### Conversion between floating-point and fixed-point

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

- **sf**: 0 1 1 1 1 0
  - **ptype**: 0 0 0 0 1 0
  - **rmode**: 0 0 0 0 0 1
  - **opcode**: 1x 0 0 0 0 0
  - **scale**: UNALLOCATED
  - -
  - **Instruction Details**: SCVTF (scalar, fixed-point) — 32-bit to single-precision
  - -
  - **Architecture Version**: FEAT_FP16
  - -
- **sf**: 0 1 1 1 1 0
  - **ptype**: 0 0 0 0 1 0
  - **rmode**: 0 0 0 0 0 1
  - **opcode**: 1x 0 0 0 0 0
  - **scale**: UNALLOCATED
  - -
  - **Instruction Details**: SCVTF (scalar, fixed-point) — 32-bit to double-precision
  - -
  - **Architecture Version**: FEAT_FP16
  - -
<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>scale</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
</table>
| 0  | 0 | 11    | 11    | 000    | FEAT_FP16
| 0  | 0 | 11    | 11    | 001    | FEAT_FP16
| 1  | 0 | 00    | 00    | 010    | -     |
| 1  | 0 | 00    | 00    | 011    | -     |
| 1  | 0 | 00    | 11    | 000    | -     |
| 1  | 0 | 00    | 11    | 001    | -     |
| 1  | 0 | 01    | 00    | 010    | -     |
| 1  | 0 | 01    | 00    | 011    | -     |
| 1  | 0 | 01    | 11    | 000    | -     |
| 1  | 0 | 01    | 11    | 001    | -     |
| 1  | 0 | 11    | 00    | 010    | FEAT_FP16
| 1  | 0 | 11    | 00    | 011    | FEAT_FP16
| 1  | 0 | 11    | 11    | 000    | FEAT_FP16
| 1  | 0 | 11    | 11    | 001    | FEAT_FP16

**Conversion between floating-point and integer**

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>ptype</th>
<th>rmode</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) — single-precision to 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) — single-precision to 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010</td>
<td>SCVTFS (scalar, integer) — 32-bit to single-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, integer) — 32-bit to single-precision</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>100</td>
<td>FCVTAS (scalar) — single-precision to 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>101</td>
<td>FCVTAVU (scalar) — single-precision to 32-bit</td>
<td></td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>Decode fields</td>
<td>ptype</td>
<td>rmode</td>
<td>opcode</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>---------------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>---------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 110</td>
<td>FMOV (general) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 00 111</td>
<td>FMOV (general) — 32-bit to single-precision</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 01 000</td>
<td>FCVTPS (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 01 001</td>
<td>FCVTPU (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1x 11x UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10 000 FCVTMS (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10 001 FCVTMU (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 000 FCVTZS (scalar, integer) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 001 FCVTZU (scalar, integer) — single-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 0x 11x UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 100 FCVTAS (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 00 101 FCVTAU (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 01 000 FCVTPS (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 01 001 FCVTPU (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 000 FCVTMS (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 001 FCVTMU (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 10 11x UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 000 FCVTZS (scalar, integer) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 001 FCVTZU (scalar, integer) — double-precision to 32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 110 FICVTZS</td>
<td></td>
<td>FEAT_JSCVT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01 11 111 UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10 11x UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 000 FCVTNS (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 001 FCVTNU (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 010 SCVTTF (scalar, integer) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 011 UCVTF (scalar, integer) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 100 FCVTAS (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 101 FCVTAU (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 110 FMOV (general) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 00 111 FMOV (general) — 32-bit to half-precision</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 01 000 FCVTPS (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 01 001 FCVTPU (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 10 000 FCVTMS (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 10 001 FCVTMU (scalar) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 11 000 FCVTZS (scalar, integer) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 11 001 FCVTZU (scalar, integer) — half-precision to 32-bit</td>
<td>FEAT_FP16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11 11 110 FCVTZS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 11x UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page 2986
<table>
<thead>
<tr>
<th>sf</th>
<th>S</th>
<th>Decode fields</th>
<th>rmode</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, integer) — 64-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, integer) — 64-bit to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>01</td>
<td>000</td>
<td>FCVTAS (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>01</td>
<td>001</td>
<td>FCVTAU (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>000</td>
<td>FCVTMS (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>001</td>
<td>FCVTMU (scalar) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>000</td>
<td>FCVTZS (scalar, integer) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>001</td>
<td>FCVTZU (scalar, integer) — single-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>x1</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, integer) — 64-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, integer) — 64-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>000</td>
<td>FCVTAS (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>001</td>
<td>FCVTAU (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>110</td>
<td>FMOV (general) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>111</td>
<td>FMOV (general) — 64-bit to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>000</td>
<td>FCVTPS (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>001</td>
<td>FCVTPU (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>1x</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>000</td>
<td>FCVTMS (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>001</td>
<td>FCVTMU (scalar) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, integer) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, integer) — double-precision to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>x0</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>01</td>
<td>110</td>
<td>FMOV (general) — top half of 128-bit to 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>01</td>
<td>111</td>
<td>FMOV (general) — 64-bit to top half of 128-bit</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
<td>1x</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, integer) — 64-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, integer) — 64-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>000</td>
<td>FCVTAS (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>001</td>
<td>FCVTAU (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>110</td>
<td>FMOV (general) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>10</td>
<td>111</td>
<td>FMOV (general) — 64-bit to half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>01</td>
<td>000</td>
<td>FCVTPS (scalar) — half-precision to 64-bit</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>
### Floating-point data-processing (1 source)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>M</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000100</td>
<td>FRINTN (scalar) — single-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000101</td>
<td>FRINTP (scalar) — single-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000111</td>
<td>FCVT — single-precision to half-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>001000</td>
<td>FRINTN (scalar) — single-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>001001</td>
<td>FRINTP (scalar) — single-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>001100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>001101</td>
<td>FCVT — single-precision to half-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>001110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>001111</td>
<td>FRINTX (scalar) — single-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>010000</td>
<td>FRINT32Z (scalar) — single-precision</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>010001</td>
<td>FRINT32X (scalar) — single-precision</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>010010</td>
<td>FRINT64Z (scalar) — single-precision</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>010011</td>
<td>FRINT64X (scalar) — single-precision</td>
<td>FEAT_FRINTTS</td>
<td></td>
</tr>
<tr>
<td>0101xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000000</td>
<td>FMOV (register) — double-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000001</td>
<td>FABS (scalar) — double-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000010</td>
<td>FNEG (scalar) — double-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000011</td>
<td>FSQRT (scalar) — double-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000100</td>
<td>FCVT — double-precision to single-precision</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>000110</td>
<td>BFCVT</td>
<td>FEAT_BF16</td>
<td></td>
</tr>
<tr>
<td>000111</td>
<td>FCVT — double-precision to half-precision</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### Floating-point compare

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001000</td>
<td>FRINTN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001001</td>
<td>FRINTP (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001010</td>
<td>FRINTM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001011</td>
<td>FRINTZ (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001100</td>
<td>FRINTA (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001110</td>
<td>FRINTX (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>001111</td>
<td>FRINTI (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010000</td>
<td>FRINT32Z (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010001</td>
<td>FRINT32X (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010010</td>
<td>FRINT64Z (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010011</td>
<td>FRINT64X (scalar) — double-precision</td>
<td>FEAT_FRINTTS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0101xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000000</td>
<td>FMOV (register) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000001</td>
<td>FABS (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000010</td>
<td>FNEG (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000011</td>
<td>FSQRT (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000100</td>
<td>FCVT — half-precision to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>000101</td>
<td>FCVT — half-precision to double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00011x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001000</td>
<td>FRINTN (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001001</td>
<td>FRINTP (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001010</td>
<td>FRINTM (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001011</td>
<td>FRINTZ (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001100</td>
<td>FRINTA (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001110</td>
<td>FRINTX (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>001111</td>
<td>FRINTI (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
Floating-point immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

Floating-point conditional compare

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.
### Floating-point data-processing (2 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

---

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FCCMP — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FCCMP — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

---

#### Floating-point data-processing (2 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>ptype</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>11</td>
<td>FMUL (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>FDIV (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>FADD (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>FSUB (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>FMAX (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>FMIN (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>FNMUL (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>FDIV (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>FADD (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>FSUB (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>FMAX (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>FMIN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FNMUL (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FDIV (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FADD (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FSUB (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FMAX (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FMIN (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>FNMUL (scalar) — half-precision</td>
<td>FEAT_FP16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
### Floating-point conditional select

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>M S ptype</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 00</td>
<td>FCSEL — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 01</td>
<td>FCSEL — double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>

### Floating-point data-processing (3 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>M S ptype o1 o0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>FMADD — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 0 0 1</td>
<td>FMSUB — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 0 1 1</td>
<td>FNMADD — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 0 1 1 0</td>
<td>FNMSUB — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 0 1 0</td>
<td>FMADD — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 1 0 1</td>
<td>FMSUB — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 1 1 1</td>
<td>FNMADD — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0 0 1 1 0 0</td>
<td>FMSUB — half-precision</td>
<td>FEAT_FP16</td>
</tr>
</tbody>
</table>

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages.

Pseudocodes

Library pseudocode for aarch32/debug/VCMatch/AArch32.VCMatch

```c
// AArch32.VCMatch()
// ==================

boolean AArch32.VCMatch(bits(32) vaddress)
if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
    // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
    match_word = Zeros(32);
    if vaddress<31:5> == ExcVectorBase()<31:5> then
        if HaveEL(EL3) && !IsSecure() then
            match_word<UInt(vaddress<4:2>) + 24> = '1';     // Non-secure vectors
        else
            match_word<UInt(vaddress<4:2>) + 0> = '1';      // Secure vectors (or no EL3)
    else
        if HaveEL(EL3) && ELUsingAArch32(EL3) && IsSecure() && vaddress<31:5> == MVBAR<31:5> then
            match_word<UInt(vaddress<4:2>) + 8> = '1';          // Monitor vectors
    // Mask out bits not corresponding to vectors.
    if !HaveEL(EL3) then
        mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
    elsif !ELUsingAArch32(EL3) then
        mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
    else
        mask = '11011110':'00000000':'11011100':'11011110';
    match_word = match_word AND DBGVCR AND mask;
    match = !IsZero(match_word);
    // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
    if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);
    if !IsZero(vaddress<1:0>) && match then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
    else
        match = FALSE;
    return match;
```

Library pseudocode for aarch32/debug/authentication/
AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```c
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ==============================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
// the state of the (DBGEN AND SPIDEN) signal.
if !HaveEL(EL3) && !IsSecure() then return FALSE;
return DBGEN == HIGH && SPIDEN == HIGH;
```
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(DBGDIR.BRPs);
    enabled = DBGCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGCR[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGCR[n].SSC, DBGCR[n].HMC, DBGCR[n].PMC,
                           linked, DBGCR[n].LBN, isbreakpnt, ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then  // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);
        if vaddress[1] == '1' && DBGCR[n].BAS == '1111' then
            // The above notwithstanding, if DBGCR[n].BAS == '1111', then it is CONSTRAINED
            // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
            // at the address DBGVR[n]+2.
            if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
            if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);
    
    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointValueMatch
The first result is whether an Address Match or Context breakpoint is programmed on the instruction at "address". The second result is whether an Address Mismatch breakpoint is programmed on the instruction, that is, whether the instruction should be stepped.

\[
\text{AArch32.BreakpointValueMatch}(\text{integer} \ n, \ \text{bits}(32) \ vaddress, \ \text{boolean} \ \text{linked}_\text{to})
\]

\[
\text{boolean,boolean} \ \text{AArch32.BreakpointValueMatch}(\text{integer} \ n, \ \text{bits}(32) \ vaddress, \ \text{boolean} \ \text{linked}_\text{to})
\]

// "n" is the identity of the breakpoint unit to match against.
// "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context matching breakpoints.
// "linked_to" is TRUE if this is a call from StateMatch for linking.

// If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
if \( n > \text{UInt}(\text{DBGDIR.BRPs}) \) then
    \( (c, n) = \text{ConstrainUnpredictableInteger}(0, \text{UInt}(\text{DBGDIR.BRPs}), \text{Unpredictable}\_\text{BPNOTIMPL}) \);
    assert \( c \in \{\text{Constraint\_DISABLED, Constraint\_UNKNOWN}\} \);
    if \( c = \text{Constraint\_DISABLED} \) then return \( (\text{FALSE,FALSE}) \);

// If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a call from StateMatch for linking).
if \( \text{DBGBCR}[n].E = '0' \) then return \( (\text{FALSE,FALSE}) \);

context_aware = \( (n \geq \text{UInt}(\text{DBGDIR.BRPs}) - \text{UInt}(\text{DBGDIR.CTX\_CMPs})) \);

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = \( \text{DBGBCR}[n].BT \);

if \( (\text{dbgtype} \in \{'011x','11xx'\} \&\& \text{!HaveVirtHostExt()} \&\& \text{!HaveV82Debug()} ) \) || // Context matching
    \( (\text{dbgtype} = '010x' \&\& \text{HaltOnBreakpointOrWatchpoint()} ) \) || // Address mismatch
    \( (\text{dbgtype} != '0x0x' \&\& \text{!context\_aware} ) \) || // Context matching
    \( (\text{dbgtype} == '1xxx' \&\& \text{!HaveEL(EL2)}) \) then // EL2 extension
    \( (c, \text{dbgtype}) = \text{ConstrainUnpredictableBits}(\text{Unpredictable}\_\text{RESBPTYPE}) \);
    assert \( c \in \{\text{Constraint\_DISABLED, Constraint\_UNKNOWN}\} \);
    if \( c = \text{Constraint\_DISABLED} \) then return \( (\text{FALSE,FALSE}) \);

// Otherwise the value returned by \text{ConstrainUnpredictableBits} must be a not-reserved value

// Determine what to compare against.
match_addr = \( (\text{dbgtype} == '0x0x') \);
mismatch = \( (\text{dbgtype} == '010x') \);
match_vmid = \( (\text{dbgtype} == '10xx') \);
match_cid1 = \( (\text{dbgtype} == 'xx1x') \);
match_cid2 = \( (\text{dbgtype} == '11xx') \);
linked = \( (\text{dbgtype} == 'xxx1') \);

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a VMID and/or context ID match, of if not context-aware. The above assertions mean that the code can just test for match\_addr == TRUE to confirm all these things.
if linked_to \&\& (!linked || match_addr) then return \( (\text{FALSE,FALSE}) \);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to \&\& linked \&\& !match\_addr then return \( (\text{FALSE,FALSE}) \);

// Do the comparison.
if match\_addr then
    byte = \( \text{UInt}(\text{vaddress}<1:0>) \);
    assert byte \( \in \{0,2\} \); // "vaddress" is halfword aligned
    byte\_select\_match = \( (\text{DBGBCR}[n].BAS<byte> == '1') \);
    integer top = 31;
    BVR\_match = \( (\text{vaddress}<\text{top}:2> = \text{DBGBVR}[n]<\text{top}:2>) \&\& \text{byte\_select\_match} \);
elsif match\_cid1 then
    BVR\_match = \( \text{PSTATE.EL} != \text{EL2} \&\& \text{CONTEXTIDR} == \text{DBGBVR}[n]<31:0>) \);
if match\_vmid then
    if \( \text{ELUsingAArch32}(\text{EL2}) \) then
        \( \text{vmid} = \text{ZeroExtend}((\text{VTTBR.VMID}, 16) , 16) \);
        \( \text{bvr\_vmid} = \text{ZeroExtend}(\text{DBGBXVR}[n]<7:0>, 16) \);
    elseif \( \text{!Have16bitVMID}() \) \&\& \( \text{VTCR\_EL2.VS == '0'} \) then
        \( \text{vmid} = \text{ZeroExtend}((\text{VTTBR\_EL2.VMID}<7:0>, 16) \);
    else
        \( \text{vmid} = \text{ZeroExtend}((\text{VTTBR\_EL2.VMID}<7:0>, 16) \);
    endif

bvr_vmid = \textbf{ZeroExtend}(\text{DBGXVR}[n]<7:0>, 16);

\text{else}

\text{vmid} = \text{VTTBR~EL2.VMID};
\text{bvr_vmid} = \text{DBGXVR}[n]<15:0>;
\text{BXVR}\_\text{match} = (\text{PSTATE.EL} \in \{\text{EL0, EL1}\} \&\& \text{EL2Enabled}() \&\& \text{vmid} == \text{bvr_vmid});

\text{elsif match\_cid2 then}

\text{BXVR}\_\text{match} = ((\text{HaveVirtHostExt}()) \| \text{HaveV82Debug}()) \&\& \text{EL2Enabled}() \&\& !\text{ELUsingAArch32}(\text{EL2}) \&\& \text{DBGXVR}[n]<31:0> == \text{CONTEXTIDR\_EL2}<31:0>);

\text{bvr\_match\_valid} = (\text{match\_addr} || \text{match\_cid1});
\text{bxvr\_match\_valid} = (\text{match\_vmid} || \text{match\_cid2});

\text{match} = (!\text{bxvr\_match\_valid} || \text{BXVR}\_\text{match}) \&\& (!\text{bvr\_match\_valid} || \text{BVR}\_\text{match});

\text{return (match \&\& !\text{mismatch,} !\text{match} \&\& \text{mismatch});}
Library pseudocode for aarch32/debug/breakpoint/AArch32.StateMatch

// AArch32.StateMatch()
// ================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
  boolean isbreakpnt, boolean ispriv)
  // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
  // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
  // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
  // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
  // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
if c == Constraint_DISABLED then return FALSE;
// Otherwise the HMC, SSC, PxC values are either valid or the values returned by
// CheckValidStateMatch are valid.

PL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
PL1_match = PxC<0> == '1';
PL0_match = PxC<1> == '1';
SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';
if !ispriv && !isbreakpnt then
  priv_match = PL0_match;
elsif SSU_match then
  priv_match = PSTATE.M IN {M32_User, M32_Svc, M32_System};
else
  case PSTATE.EL of
    when EL3 priv_match = PL1_match; // EL3 and EL1 are both PL1
    when EL2 priv_match = PL2_match;
    when EL1 priv_match = PL1_match;
    when EL0 priv_match = PL0_match;
  end_case

  case SSC of
    when '00' security_state_match = TRUE; // Both
    when '01' security_state_match = !IsSecure(); // Non-secure only
    when '10' security_state_match = IsSecure(); // Secure only
    when '11' security_state_match = (HMC == '1' || IsSecure()); // HMC=1 -> Both, 0 -> Secure only
  end_case

if linked then
  // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
  // it is CONstrained UNpredictable whether this gives no match, or LBN is mapped to some
  // UNKNOWN breakpoint that is context-aware.
  lbnum = UInt(LBN);
  first_ctx_cmp = (UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
  last_ctx_cmp = UInt(DBGDIDR.BRPs);
  if (lbnum < first_ctx_cmp || lbnum > last_ctx_cmp) then
    (c, lbnum) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTXCA);
    assert c IN (ConstraintDISABLED, ConstraintNONE, ConstraintUNKNOWN);
    case c of
      when ConstraintDISABLED return FALSE; // Disabled
      when ConstraintNONE linked = FALSE; // No linking
      // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
    end_case

if linked then
  vaddress = bits(32) UNKNOWN;
  linked_to = TRUE;
  (linked_match, -) = AArch32.BreakpointValueMatch(lbnum, vaddress, linked_to);
return priv_match && security_state_match && (!linked || linked_match);
// AArch32.GenerateDebugExceptions()
// --------------------------------

boolean AArch32.GenerateDebugExceptions()
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure());

// AArch32.GenerateDebugExceptionsFrom()
// -------------------------------------

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)
if from == EL0 && (ELIsInHost(from) || !ELStateUsingAArch32(EL1, secure)) then
    mask = bit UNKNOWN;  // PSTATE.D mask, unused for EL0 case
    return AArch64.GenerateDebugExceptionsFrom(from, secure, mask);
if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
    return FALSE;
if HaveEL(EL3) && secure then
    assert from != EL2;  // Secure EL2 always uses AArch64
    if IsSecureEL2Enabled() then
        // Implies that EL3 and EL2 both using AArch64
        enabled = MDCR_EL3.SDD == '0';
    else
        spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
        if spd<1> == '1' then
            enabled = spd<0> == '1';
        else
            // SPD == 0b01 is reserved, but behaves the same as 0b00.
            // Implies that EL3 and EL2 both using AArch64
            enabled = from != EL2;    // Secure EL2 always uses AArch64
        else
            enabled = EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from != EL2;
    return enabled;

// AArch32.CheckForPMUOverflow()
// -----------------------------

boolean AArch32.CheckForPMUOverflow()
if !ELUsingAArch32(EL1) then return AArch64.CheckForPMUOverflow();
    pmuirq = PMCR.E == '1' && PMINTENSET<31> == '1' && PMOVSET<31> == '1';
    for n = 0 to Uint(PMCR.N) - 1
        if HaveEL(EL2) then
            hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
            hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
            E = (if n < Uint(hpmn) then PMCR.E else hpme);
        else
            E = PMCR.E;
        if E == '1' && PMINTENSET<n> == '1' && PMOVSET<n> == '1' then pmuirq = TRUE;
    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)
    return pmuirq;
Library pseudocode for aarch32/debug/pmu/AArch32.CountEvents
boolean AArch32.CountEvents(integer n)
assert n == 31 || n < UInt(PMCR.N);

if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);

// In Non-secure state, some counters are reserved for EL2
if HaveEL(EL2) then
    hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
    hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
    resvd_for_el2 = n >= UInt(hpmn) && n != 31;
else
    resvd_for_el2 = FALSE;

// Main enable controls
if resvd_for_el2 then
    E = if ELUsingAArch32(EL2) then HDCR.HPME else MDCR_EL2.HPME;
else
    E = PMCR.E;
enabled = E == '1' && PMCNTENSET<n> == '1';

// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;

// Event counting in Secure state is prohibited if all of:
// * EL3 is implemented
// * One of the following is true:
//   * EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
//     * FEAT_PMUv3p7 is not implemented
//     * MDCR_EL3.MMPX == 0
//   * EL3 is using AArch32 and SDCR.SPME == 0
//   * Not executing at EL0, or SDER.SUNIDEN == 0
if HaveEL(EL3) && IsSecure() then
    spme = if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
    if !ELUsingAArch32(EL3) && HavePMUv3p7() then
        prohibited = spme == '0' && MDCR_EL3.MMPX == '0';
    else
        prohibited = spme == '0';
    if prohibited && PSTATE.EL == EL0 then
        prohibited = SDER.SUNIDEN == '0';

// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * HDCR.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
    prohibited = HDCR.HPMD == '1';

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUEnableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// PMCR.DP disables the cycle counter when event counting is prohibited
if enabled && prohibited && n == 31 then
    enabled = PMCR.DP == '0';

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR.DP.
if Havev85PMU() && n == 31 then
    if HaveEL(EL3) && IsSecure() then
        sccd = if ELUsingAArch32(EL3) then SDCR.SCCD else MDCR_EL3.SCCD;
        if sccd == '1' then prohibited = TRUE;
    if PSTATE.EL == EL2 && HDCR.HCCD == '1' then
        prohibited = TRUE;
// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() & & n != 31 then
    ovflw = PMOVSR<UInt>(PMCR.N)-1:0;
    if resvd_for_el2 then
        FZ = if ELUsingAArch32(EL2) then HDCR.HPMFZO else MDCR_EL2.HPMFZO;
        ovflw<UInt>(hpnm)-1:0> = Zeros();
    else
        FZ = PMCR.FZO;
        if HaveEL(EL2) then
            ovflw<UInt>(PMCR.N)-1:UInt>(hpmn)> = Zeros();
        else
            FZ = PMCR.FZO;
    frozen = FZ == '1' & & !IsZero(ovflw);

// Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
filter = if n == 31 then PMCCFILTR else PMEVTYPER[n];
P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
case PSTATE.EL of
    when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
    when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
    when EL2 filtered = NSH == '0';
    when EL3 filtered = P == '1';
return !debug & & enabled & & !prohibited & & !filtered & & !frozen;

// Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

// Take an exception in Debug state to Hyp mode.
AArch32.EnterHypModeInDebugState(ExceptionRecord exception);
SynchronizeContext();
assert HaveEL(EL2) & & !IsSecure() & & ELUsingAArch32(EL2);
AArch32.ReportHypEntry(exception);
AArch32.WriteMode(M32_Hyp);
SPSR[] = bits(32) UNKNOWN;
ELR_hyp = bits(32) UNKNOWN;
// In Debug state, the PE always execute T32 instructions when in AArch32 state, and
// PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
PSTATE.T = '1'; // PSTATE.J is RES0
PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
DLR = bits(32) UNKNOWN;
DSPSR = bits(32) UNKNOWN;
PSTATE.E = HSCTRL.EE;
PSTATE.I = '0';
PSTATE.IT = '00000000';
if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
EDSCR.ERR = '1';
UpdateEDSCRFields();
EndOfInstruction();
// AArch32.EnterModeInDebugState()
// ========================================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                  // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.I = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    ESCR.ERR = '1';
    UpdateEDSCRFields();           // Update EDSCR processor state flags.
    EndOfInstruction();

// AArch32.EnterMonitorModeInDebugState()
// ========================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                  // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.I = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        DLR = bits(32) UNKNOWN;
        DSPSR = bits(32) UNKNOWN;
        ESCR.ERR = '1';
        UpdateEDSCRFields();           // Update EDSCR processor state flags.
    EndOfInstruction();
// AArch32.WatchpointByteMatch()
// -----------------------------------------------

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
{
    integer top = 31;
    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;  // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt>(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    if DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '1111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASECONTIGUOUS);
    bottom = 3;  // For the whole doubleword

    if the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 & mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESPWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE;  // Disabled
            when Constraint_NONE mask = 0;  // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        // If the DBGxVR<n> EL1.RESS field bits are not a sign extension of the MSB
        // of DBGBVR<n> EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
            if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                top = 63;
            WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMAKEDBITS);
        else
            WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;

    return WVR_match && byte_select_match;
}
// Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv, AccType acctype, boolean iswrite)

assert ELUsingAArch32(S1TranslationRegime());
assert n <= UInt(DBGDIDR.WRPs);

"ispriv" is:
// * FALSE for all loads, stores, and atomic operations executed at EL0.
// * FALSE if the access is unprivileged.
// * TRUE for all other loads, stores, and atomic operations.

enabled = DBGWCR[n].E == '1';
linked = DBGWCR[n].WT == '1';
isbreakpnt = FALSE;

state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC, linked, DBGWCR[n].LBN, isbreakpnt, ispriv);

ls_match = FALSE;
ls_match = (DBGWCR[n].LSC<(if iswrite then 1 else 0)> == '1');

value_match = FALSE;
for byte = 0 to size - 1
  value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);
return value_match && state_match && ls_match && enabled;

// Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
  route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
    (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
    IsDebugException(fault) && MDCR_EL2.TDE == '1');

if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
  route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

if route_to_aarch64 then
  AArch64.Abort(ZeroExtend(vaddress), fault);
elsif fault.acctype == AccType_IFETCH then
  AArch32.TakePrefetchAbortException(vaddress, fault);
else
  AArch32.TakeDataAbortException(vaddress, fault);
Library pseudocode for aarch32/exceptions/aborts/AArch32.AbandonSyndrome

// AArch32.AbandonSyndrome()
// =======================
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbandonSyndrome(Exception exceptype, FaultRecord fault, bits(32) vaddress)
exception = ExceptionSyndrome(exceptype);
d_side = exceptype == Exception_DataAbort;
exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
exception.vaddress = ZeroExtend(vaddress);
if IPAValid(fault) then
    exception.ipavalid = TRUE;
    exception.NS = fault.ipaddress.NS;
    exception.ipaddress = ZeroExtend(fault.ipaddress.address);
else
    exception.ipavalid = FALSE;
return exception;

Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()

bits(32) pc = ThisInstrAddr();
if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
    if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

    // Generate an Alignment fault Prefetch Abort exception
    vaddress = pc;
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    secondstage = FALSE;
    AArch32.Abort(vaddress, AArch32.AlignmentFault(acctype, iswrite, secondstage));
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)

// The encoding used in the IFSR or DFSR can be Long-descriptor format or Short-descriptor format. Normally, the current translation table format determines the format. For an abort from Non-secure state to Monitor mode, the IFSR or DFSR uses the Long-descriptor format if any of the following applies:
// * The Secure TTBCR.EAE is set to 1.
// * The abort is synchronous and either:
//   - It is taken from Hyp mode.
//   - It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
long_format = FALSE;
if route_to_monitor && !IsSecure() then
  long_format = TTBCR_S.EAE == '1';
else
  long_format = TTBCR.EAE == '1';
d_side = TRUE;
if long_format then
  syndrome = AArch32.FaultStatusLD(d_side, fault);
else
  syndrome = AArch32.FaultStatusSD(d_side, fault);
if fault.acctype == AccType_IC then
  if (!long_format && boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
    i_syndrome = syndrome;
    syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
  else
    i_syndrome = bits(32) UNKNOWN;
  if route_to_monitor then
    IFSR_S = i_syndrome;
  else
    IFSR = i_syndrome;
else
  if route_to_monitor then
    DFSR_S = syndrome;
    DFAR_S = vaddress;
  else
    DFSR = syndrome;
    DFAR = vaddress;
return;
Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

// AArch32.ReportPrefetchAbort()
// =============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && !IsSecure() then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    d_side = FALSE;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

// AArch32.TakeDataAbortException()
// ================================

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (HaveEL(EL2) && !IsSecure() && PSTATE.EL IN {EL0, EL1}) &&
        (HCR.TGE == '1' || IsSecondStage(fault) ||
        HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1'));

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        end if
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
route_to_hyp = (HaveEL(EL2) && IsSecure() && PSTATE.EL IN {EL0, EL1}) &&
(HCR.TGE == '1' || IsSecondStage(fault) ||
(HaveRASExt() && HCR.TEA == '1' && IsExternalAbort(fault)) ||
(IsDebugException(fault) && HCR.TDE == '1'));

bits(32) preferred_exception_return = ThisInstrAddr();

vect_offset = 0x0C;
lr_offset = 4;
if IsDebugException(fault) then DBGDSRec.MOE = fault.debugmoe;
if route_to_monitor then
  AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
  AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
  if fault.statuscode == Fault_Alignment then // PC Alignment fault
    exception = ExceptionSyndrome(Exception_PCAlignment);
    exception.vaddress = ThisInstrAddr();
  else
    exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
    if PSTATE.EL == EL2 then
      AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
      AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
  else
    AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
else
  AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
  AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

AArch32.TakePhysicalFIQException()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
  route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
  route_to_aarch64 = SCR_EL3.FIQ == '1';
if route_to_aarch64 then
  AArch64.TakePhysicalFIQException();
route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
  (HCR.TGE == '1' || HCR.FMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x1C;
lr_offset = 4;
if route_to_monitor then
  AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
  exception = ExceptionSyndrome(Exception_FIQ);
  AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
  AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
AArch32.TakePhysicalIRQException()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
    route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.IRQ == '1';
if route_to_aarch64 then AArch64.TakePhysicalIRQException();

route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
    (HCR.TGE == '1' || HCR.IMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x18;
lr_offset = 4;
if route_to_monitor then
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elsif PSTATE.EL == EL2 || route_to_hyp then
    exception = ExceptionSyndrome(Exception_IRQ);
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/asynch/AArch32.TakePhysicalSErrorException

// AArch32.TakePhysicalSErrorException()
// -----------------------------------

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) errortype,
                                boolean impdef_syndrome, bits(24) full_syndrome)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 &&
        route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));

    if !route_to_aarch64 &&
        HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(impdef_syndrome, full_syndrome);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                     (HCR.TGE == '1' || HCR.AMO == '1'));

    bits(32) preferred_exception_return = ThisInstrAddr();
    lr_offset = 8;

    if IsSErrorEdgeTriggered(full_syndrome) then
        ClearPendingPhysicalSError();

    fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
    vaddress = bits(32) UNKNOWN;

    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        AArch32.AbortSyndrome(Expect_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        end
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualFIQException

// AArch32.TakeVirtualFIQException()
// -----------------------------------

AArch32.TakeVirtualFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualIRQException

// AArch32.TakeVirtualIRQException()
// --------------------------------

AArch32.TakeVirtualIRQException()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
if ELUsingAArch32(EL2) then // Virtual IRQs enabled if TGE==0 and IM0==1
    assert HCR.TGE == '0' && HCR.IM0 == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.IM0 == '1';
// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x18;
lr_offset = 4;
AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);

Library pseudocode for aarch32/exceptions/asynch/AArch32.TakeVirtualSErrorException

// AArch32.TakeVirtualSErrorException()
// -----------------------------------

AArch32.TakeVirtualSErrorException(bit extflag, bits(2) errortype, boolean impdef_syndrome, bits(24) full_syndrome)
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
if ELUsingAArch32(EL2) then // Virtual SError enabled if TGE==0 and AMO==1
    assert HCR.TGE == '0' && HCR.AMO == '1';
else
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
// Check if routed to AArch64 state
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSErrorException(impdef_syndrome, full_syndrome);
route_to_monitor = FALSE;
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x10;
lr_offset = 8;
vaddress = bits(32) UNKNOWN;
parity = FALSE;
if HaveRASExt() then
    if ELUsingAArch32(EL2) then
        fault = AArch32.AsynchExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
    else
        fault = AArch32.AsynchExternalAbort(FALSE, VSESR_EL2.AET, VSESR_EL2.ExT);
    else
        fault = AArch32.AsynchExternalAbort(parity, errortype, extflag);
ClearPendingVirtualSError();
AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```c
// AArch32.SoftwareBreakpoint()
// ============================
AArch32.SoftwareBreakpoint(bits(16) immediate)

if (EL2Enabled() && !ELUsingAArch32(EL2) &&
    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1') || !ELUsingAArch32(EL1) then
    AArch64.SoftwareBreakpoint(immediate);

vaddress = bits(32) UNKNOWN;
acctype = AccType_IFETCH;           // Take as a Prefetch Abort
iswrite = FALSE;
entry = DebugException_BKPT;

fault = AArch32.DebugFault(acctype, iswrite, entry);
AArch32.Abort(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/debug/DebugException

```c
constant bits(4) DebugException_Breakpoint  = '0001';
constant bits(4) DebugException_BKPT        = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint  = '1010';
```

Library pseudocode for aarch32/exceptions/exceptions/
AArch32.CheckAdvSIMDOrFPRegisterTraps

```c
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =======================================
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.
AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

if PSTATE.EL == EL1 && EL2Enabled() then
    tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
    tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

if (tid0 == '1' && reg == '0000')                             // FPSID
    || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then    // MVFRx
    if ELUsingAArch32(EL2) then
        AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
        AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
```
Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer, bit) AArch32.ExceptionClass(Exception exc) {
    il_is_valid = TRUE;

    case exc of
        when Exception_Uncategorized    ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap        ec = 0x01;
        when Exception_CP15RRTTrap     ec = 0x03;
        when Exception_CP14RTTrap      ec = 0x05;
        when Exception_CP14DTTrap      ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap ec = 0x07;
        when Exception_FPIDTrap       ec = 0x08;
        when Exception_PACTrap        ec = 0x09;
        when Exception_LDST64BTrap     ec = 0x0A;
        when Exception_TSTARTAccessTrap ec = 0x1B;
        when Exception_CP14RRTTrap     ec = 0x0C;
        when Exception_BranchTarget   ec = 0x0D;
        when Exception_IllegalState   ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall ec = 0x11;
        when Exception_HypervisorCall ec = 0x12;
        when Exception_MonitorCall    ec = 0x13;
        when Exception_InstructionAbort ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAlignment    ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort      ec = 0x24;
        when Exception_NV2DataAbort    ec = 0x25;
        when Exception_FPTrappedException ec = 0x28;
        otherwise           Unreachable();

        if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
            ec = ec + 1;

        if il_is_valid then
            il = if ThisInstrLength() == 32 then '1' else '0';
        else
            il = '1';

        return (ec,il);
}

Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64() {
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1');
}
```c
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)
{
    Exception exctype = exception.exctype;
    (ec,il) = AArch32.ExceptionClass(exctype);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';
    HSR = ec<5:0>:il:iss;

    if exctype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif exctype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;
    else
        if exception.ipavvalid then
            HPFAR<31:4> = exception.ipaddress<39:12>;
            else
                HPFAR<31:4> = bits(28) UNKNOWN;
        return;
```
// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert HighestELUsingAArch32();
    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                     // Secure state
    elseif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);
    // Reset the CP14 and CP15 registers and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';
    // Reset all other PSTATE fields, including instruction set andendianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111';       // All asynchronous exceptions masked
    PSTATE.IT = '00000000';       // IT block state reset
    PSTATE.T = SCTLR.TE;          // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
    PSTATE.E = SCTLR.EE;          // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0';              // Clear Illegal Execution state bit
    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);
    bits(32) rv;                      // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        if MVBAR<0> == '1' then       // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
        else
            rv = RVBAR<31:1>:'0';
        // The reset vector must be correctly aligned
        assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');
        BranchTo(rv, BranchType_RESET);

Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

// AArch32.FPTrappedException()
// ============================================

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
        FPEXC.DEX = '1';
        FPEXC.TFV = '1';
        FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;                      // IDF,IXF,UFF,OFF,DZF,IOF
        FPEXC<10:8> = '111';                                              // VECITR is RES1
    AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

// AArch32.CallHypervisor()
// -----------------------------
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);
    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

// AArch32.CallSupervisor()
// -----------------------------
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate)
    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCException

// AArch32.TakeHVCException()
// -----------------------------

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
#include <lib/aarch32.h>
#include <lib/aarch32.h>

// AArch32.TakeSMCException()
// ================
AArch32.TakeSMCException()
assert HaveEL(EL3) & ELUsingAAArch32(EL3);
AArch32.ITAdvance();
SSAdvance();
bv(32) preferred_exception_return = NextInstrAddr();
vect_offset = 0x08;
lr_offset = 0;
AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);

// AArch32.TakeSVCException(bits(16) immediate)
// ================
AArch32.TakeSVCException(bits(16) immediate)
AArch32.ITAdvance();
SSAdvance();
rte_to_hyp = PSTATE.EL == EL0 & EL2Enabled() & HCR.TGE == '1';

bv(32) preferred_exception_return = NextInstrAddr();
vect_offset = 0x08;
lr_offset = 0;

if PSTATE.EL == EL2 || rte_to_hyp then
    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
    AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return, integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) & !IsSecure() & ELUsingAArch32(EL2);
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32_ReportHypEntry(exception);
    AArch32_WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTRL.TE;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32_ReportHypEntry(exception);
    AArch32_WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTRL.TE;
    ShouldAdvanceSS = FALSE;
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTRL.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
    EndOfInstruction();

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset, integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32_WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;
    ShouldAdvanceSS = FALSE;
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
        PSTATE.E = SCTLR.EE;
        PSTATE.IL = '0';
        PSTATE.IT = '00000000';
        if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
        if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
        BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
    EndOfInstruction();

Shared Pseudocode Functions
// AArch32.EnterMonitorMode()
// ================
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset, 
integer vect_offset)

  SynchronizeContext();
  assert HaveEL(EL3) & ELLUsingAArch32(EL3);
  from_secure = IsSecure();
  bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
  if PSTATE.M == M32_Monitor then SCR.NS = '0';
  AArch32.WriteMode(M32_Monitor);
  SPSR[] = spsr;
  R[14] = preferred_exception_return + lr_offset;
  PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
  PSTATE.SS = '0';
  ShouldAdvanceSS = FALSE;
  PSTATE.<A,I,F> = '111';
  PSTATE.E = SCTLR.EE;
  PSTATE.IL = '0';
  PSTATE.IT = '00000000';
  if HavePANExt() then
    if !from_secure then
      PSTATE.PAN = '0';
    elsif SCTLR.SPAN == '0' then
      PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION);
  EndOfInstruction();

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)

if PSTATE.EL == EL0 && (!HaveEL(EL2) || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0')) && !ELUsingAArch32(EL1)
    AArch64.CheckFPAdvSIMDEnabled();
else if PSTATE.EL == EL0 && HaveEL(EL2) && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && !ELUsingAArch32(EL1)
    if fpexc_check && HCR_EL2.RW == '0' then
        fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
        if fpexc_en == '0' then UNDEFINED;
        AArch64.CheckFPAdvSIMDEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;
        if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';
        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD enabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;
            // Check if access disabled in CPACR
            case cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '10' disabled = ConstrnUnpredictableBool(Unpredictable_RESCPACR);
                when '11' disabled = FALSE;
                if disabled then UNDEFINED;
            if required, check FPEXC enabled bit.
            if fpexc_check && FPEXC.EN == '0' then UNDEFINED;
            AArch32.CheckFPAdvSIMDTrap(advsimd);  // Also check against HCPTR and CPTR_EL3
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
  if EL2Enabled() && !ELUsingAArch32(EL2) then
    AArch64.CheckFPAdvSIMDTrap();
  else
    if HaveEL(EL2) && !IsSecure() then
      hcptr_tase = HCPR.TASE;
      hcptr_cp10 = HCPR.TCP10;
    end
    if HaveEL(EL3) && ELUsingAArch32(EL3) && !IsSecure() then
      // Check if access disabled in NSACR
      if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
      if NSACR.cp10 == '0' then hcptr_cp10 = '1';
    end
    // Check if access disabled in HCPR
    if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
      exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
      exception.syndrome<24:20> = ConditionSyndrome();
      if advsimd then
        exception.syndrome<5> = '1';
      else
        exception.syndrome<5> = '0';
      end
      exception.syndrome<3:0> = '1010';         // coproc field, always 0xA
    end
    if PSTATE.EL == EL2 then
      AArch32.TakeUndefInstrException(exception);
    else
      AArch32.TakeHypTrapException(exception);
    end
  end
  if HaveEL(EL3) && !ELUsingAArch32(EL3) then
    // Check if access disabled in CPTR_EL3
    if CPTR_EL3.TFP == '1' then
      AArch64.AdvSIMDFPAccessTrap(EL3);
    end
  end
return;

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

// AArch32.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
  if !HaveEL(EL3) || PSTATE.EL == EL0 then
    UNDEFINED;
  if EL2Enabled() && !ELUsingAArch32(EL2) then
    AArch64.CheckForSMCUndefOrTrap(Zeros(16));
  else
    route_to_hyp = HaveEL(EL2) && !IsSecure() && PSTATE.EL == EL1 && HCR.TSC == '1';
    if route_to_hyp then
      exception = ExceptionSyndrome(Exception_MonitorCall);
      AArch32.TakeHypTrapException(exception);
    end
return;
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if HaveFGEExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAAArch32(EL1) && E2LEnabled() && HFGITR.EL2.SVC_EL0 == '1' &&
                (HCR_EL2.<E2H, TGE> != '1' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));
    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWFxTrap

// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);
    // Check for routing to AArch64
    if !ELUsingAAArch32(target_el) then
        AArch64.CheckForWFxTrap(target_el, is_wfe);
        return;
    case target_el of
        when EL1
            trap = (is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2
            trap = (is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3
            trap = (is_wfe then SCR.TWE else SCR.TWI) == '1';
    if trap then
        if target_el == EL1 && E2LEnabled() && !ELUsingAAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WFxTrap(target_el, is_wfe);
        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            exception = ExceptionSyndrome(Exception_WFxTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
            exception.syndrome<0> = if is_wfe then '1' else '0';
            AArch32.TakeHypTrapException(exception);
        else
            AArch32.TakeUndefInstrException();
    else
        AArch64.WFxTrap(target_el, is_wfe);
    end case;
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;
        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle(NextInstrAddr(), 2, AccType_IFETCH, TRUE);
        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
            '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            UNDEFINED;
        return;

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;
        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode (exception, preferred_exception_return, 0x14);
            else
                AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

// AArch32.CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
    if setend_disabled == '1' then
        UNDEFINED;
    return;
// AArch32.SystemAccessTrap()
// =================================
// Trapped system register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
(valid, target_el) = ELFromM32(mode);
assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

if target_el == EL2 then
    exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(exception);
else
    AArch32.TakeUndefInstrException();
// Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome
// ----------------------------------
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;
    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise    Unreachable();
    endcase;

    bits(20) iss = Zerps();
    if exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss<13:10> = instr<19:16>; // CRn, Reg in case of VMRS
        iss<8:5>  = instr<15:12>;  // Rt
        iss<9>    = '0';           // RES0
        if exception.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>; // opc2
            iss<16:14> = instr<23:21>; // opc1
            iss<4:1>  = instr<3:0>;   // CRm
        else //VMRS Access
            iss<19:17> = '000';       //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';       //opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';      //CRm  - Hardcoded for VMRS
        endif;
    elseif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;  // opc1
        iss<13:10> = instr<19:16>; // Rt2
        iss<8:5>  = instr<15:12>;  // Rt
        iss<4:1>  = instr<3:0>;   // CRm
    elseif exception.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;  // imm8
        iss<4>    = instr<23>;     // U
        iss<2:1>  = instr<24,21>;  // P,W
        if instr<19:16> == '1111' then // Rn==15, LDC(Literal addressing)/STC
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>  = '1';
        elseif exception.exceptype == Exception_Uncategorized then
            // Trapped for unknown reason
            iss<8:5> = instr<19:16>;  // Rn
            iss<3>  = '0';
        endif;
    endif;

    iss<0> = instr<20>;       // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0> = iss;
    return exception;
Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```c
// AArch32.TakeHypTrapException()
// ========================================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(exception);

// AArch32.TakeHypTrapException()
// ========================================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2);
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x14;
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```c
// AArch32.TakeMonitorTrapException()
// ========================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```c
// AArch32.TakeUndefInstrException()
// ========================================

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// ========================================

AArch32.TakeUndefInstrException(ExceptionRecord exception)
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
```
Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

// AArch32.UndefineFault()
// ========================

AArch32.UndefineFault()

  if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefineFault();
  AArch32.TakeUndefInstrException();

Library pseudocode for aarch32/functions/aborts/AArch32.CreateFaultRecord

// AArch32.CreateFaultRecord()
// ===========================

FaultRecord AArch32.CreateFaultRecord(Fault statuscode, bits(40) ipaddress, bits(4) domain, integer level, AccType acctype, boolean write, bit extflag, bits(4) debugmoe, bits(2) errortype, boolean secondstage, boolean s2fs1walk)

  FaultRecord fault;
  fault.statuscode = statuscode;
  if (statuscode != Fault_None && PSTATE.EL != EL2 && TTBCR.EAE == '0' && !secondstage && !s2fs1walk && AArch32.DomainValid(statuscode, level)) then
    fault.domain = domain;
  else
    fault.domain = bits(4) UNKNOWN;
  fault.errortype = errortype;
  fault.ipaddress.NS = bit UNKNOWN;
  fault.ipaddress.address = ZeroExtend(ipaddress);
  fault.level = level;
  fault.acctype = acctype;
  fault.write = write;
  fault.extflag = extflag;
  fault.secondstage = secondstage;
  fault.s2fs1walk = s2fs1walk;

  return fault;

Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

// AArch32.DomainValid()
// =====================

// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
  assert statuscode != Fault_None;

  case statuscode of
    when Fault_Domain
      return TRUE;
    when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
      return level == 2;
    otherwise
      return FALSE;
Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusLD

// AArch32.FaultStatusLD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            if fault.write then '1' else '0';
        if IsExternalAbort(fault) then fsr<12> = fault.extflag;
        fsr<9> = '1';
        fsr<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
    return fsr;

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusSD

// AArch32.FaultStatusSD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            if fault.write then '1' else '0';
        if IsExternalAbort(fault) then fsr<12> = fault.extflag;
        fsr<9> = '0';
        fsr<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);
    if d_side then
        fsr<7:4> = fault.domain;               // Domain field (data fault only)
    return fsr;
Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

// AArch32.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;
    bits(25) iss = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then
        iss<11:10> = fault.errortype; // AET
    if d_side then
        if ( IsSecondStage(fault) && !fault.s2fs1walk && (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.acctype == AccType_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")) ) then
            iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
    return iss;
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

bits(5) result;
case statuscode of
    when Fault_AccessFlag
        assert level IN {1,2};
        result = if level == 1 then '00011' else '00110';
    when Fault_Altignment
        result = '00001';
    when Fault_Permission
        assert level IN {1,2};
        result = if level == 1 then '01101' else '01111';
    when Fault_Domain
        assert level IN {1,2};
        result = if level == 1 then '01001' else '01011';
    when Fault_Translation
        assert level IN {1,2};
        result = if level == 1 then '00101' else '00111';
    when Fault_SyncExternal
        result = '01000';
    when Fault_SyncExternalOnWalk
        assert level IN {1,2};
        result = if level == 1 then '01100' else '01110';
    when Fault_SyncParity
        result = '11001';
    when Fault_SyncParityOnWalk
        assert level IN {1,2};
        result = if level == 1 then '11100' else '11110';
    when Fault_AsyncParity
        result = '11000';
    when Fault_AsyncExternal
        result = '10110';
    when Fault_Debug
        result = '00010';
    when Fault_TLBConflict
        result = '10000';
    when Fault_Lockdown
        result = '10100';  // IMPLEMENTATION DEFINED
    when Fault_Exclusive
        result = '10101';  // IMPLEMENTATION DEFINED
    when Fault_ICacheMaint
        result = '00100';
    otherwise
        Unreachable();

return result;

// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

// PSTATE.C argument to following function call does not affect the imm32 result.
(imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

return imm32;
Library pseudocode for aarch32/functions/common/A32ExpandImm_C

// A32ExpandImm_C()
// ================
(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

  unrotated_value = ZeroExtend(imm12<7:0>, 32);
  (imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UInt(imm12<11:8>), carry_in);
  return (imm32, carry_out);

Library pseudocode for aarch32/functions/common/DecodeImmShift

// DecodeImmShift()
// ================
(SRTYPE, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

  case srtype of
    when '00'
      shift_t = SRTYPE_LSL; shift_n = UInt(imm5);
    when '01'
      shift_t = SRTYPE_LSR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
    when '10'
      shift_t = SRTYPE_ASR; shift_n = if imm5 == '00000' then 32 else UInt(imm5);
    when '11'
      if imm5 == '00000' then
        shift_t = SRTYPE_RRX; shift_n = 1;
      else
        shift_t = SRTYPE_ROR; shift_n = UInt(imm5);
  return (shift_t, shift_n);

Library pseudocode for aarch32/functions/common/DecodeRegShift

// DecodeRegShift()
// ================
SRTYPE DecodeRegShift(bits(2) srtype)

  case srtype of
    when '00'  shift_t = SRTYPE_LSL;
    when '01'  shift_t = SRTYPE_LSR;
    when '10'  shift_t = SRTYPE_ASR;
    when '11'  shift_t = SRTYPE_ROR;
  return shift_t;

Library pseudocode for aarch32/functions/common/RRX

// RRX()
// ======

<bits(N) RRX(bits(N) x, bit carry_in)
  (result, -) = RRX_C(x, carry_in);
  return result;

Library pseudocode for aarch32/functions/common/RRX_C

// RRX_C()
// ========
<bits(N), bit) RRX_C(bits(N) x, bit carry_in)
  result = carry_in : x<N-1:1>;
  carry_out = x<0>;
  return (result, carry_out);
**Library pseudocode for aarch32/functions/common/SRType**

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};

**Library pseudocode for aarch32/functions/common/Shift**

```plaintext
// Shift()
// ========

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift_C(value, srtype, amount, carry_in);
    return result;
```

**Library pseudocode for aarch32/functions/common/Shift_C**

```plaintext
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType_RRX && amount != 1);
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);
        return (result, carry_out);
```

**Library pseudocode for aarch32/functions/common/T32ExpandImm**

```plaintext
// T32ExpandImm()
// =============

bits(32) T32ExpandImm(bits(12) imm12)
    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);
    return imm32;
```
Library pseudocode for aarch32/functions/common/T32ExpandImm_C

// T32ExpandImm_C()
// ================
(void) T32ExpandImm_C(bits(12) imm12, bit carry_in)

if imm12<11:10> == '00' then
  case imm12<9:8> of
    when '00' then
      imm32 = ZeroExtend(imm12<7:0>, 32);
    when '01' then
      imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
    when '10' then
      imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
    when '11' then
      imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
  end_case;
  carry_out = carry_in;
else
  unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
  (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));
end_if;
return (imm32, carry_out);

Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

// AArch32.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained CP15 traps in HSTR and HCR.

boolean AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

  if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
    if PSTATE.EL == EL0 && !(ELUsingAArch32(EL2)) then
      return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);
  end_if;
  // Check for MCR, MRC, MCRR and MRRC disabled by HSTR<CRn/CRm>
  major = if nreg == 1 then CRn else CRm;
  if !(major IN {4,14}) && HSTR<major> == '1' then
    return TRUE;
  end_if;
  // Check for MRC and MCR disabled by HCR.TIDCP
  if (HCR.TIDCP == '1' && nreg == 1 &&
    ((CRn == 9 && CRm IN {0,1,2,5,6,7,8}) ||
     (CRn == 10 && CRm IN {0,1,4,8}) ||
     (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15})) then
    return TRUE;
  end_if;
return FALSE;
// AArch32.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AArch32.AccType_ATOMIC;
    iswrite = TRUE;
    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;

Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
// It is always safe to return TRUE which will check the physical address only.

boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);

Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.

AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
// AArch32.SetExclusiveMonitors()
// ==============================

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
  acctype = AccType_ATOMIC;
  iswrite = FALSE;
  aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
  memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    return;
  if memaddrdesc.memattrs.shareable then
    MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    AArch32.MarkExclusiveVA(address, ProcessorID(), size);

Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

// CheckAdvSIMDEnabled()
// ---------------------

CheckAdvSIMDEnabled()
  fpexc_check = TRUE;
  advsimd = TRUE;
  AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
  // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted
  // Make temporary copy of D registers
  // _Dclone[] is used as input data for instruction pseudocode
  for i = 0 to 31
    _Dclone[i] = D[i];
  return;

Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

// CheckAdvSIMDOrVFPEnabled()
// --------------------------

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
  AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
  // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
  return;

Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

// CheckCryptoEnabled32()
// ----------------------

CheckCryptoEnabled32()
  CheckAdvSIMDEnabled();
  // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
  return;
Library pseudocode for aarch32/functions/float/CheckVFPEnabled

// CheckVFPEnabled()
// ================

CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;

Library pseudocode for aarch32/functions/float/FPHalvedSub

// FPHalvedSub()
// =============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPIInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPIInfinity('1');
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
            return result;

Library pseudocode for aarch32/functions/float/FPRSqrtStep

// FPRSqrtStep()
// ==============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0');
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0');
        result = FPHalvedSub(three, product, fpcr);
    return result;
Library pseudocode for aarch32/functions/float/FPRecipStep

// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
assert N IN {16,32};
FPCRType fpcr = StandardFPSCRValue();
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    inf1 = (type1 == FPType_Infinity);
    inf2 = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);  
    zero2 = (type2 == FPType_Zero);
    bits(N) product;
    if (inf1 && zero2) || (zero1 && inf2) then
        product = FPZero('0');
    else
        product = FPMul(op1, op2, fpcr);
    bits(N) two = FPTwo('0');
    result = FPSub(two, product, fpcr);
return result;

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

// StandardFPSCRValue()
// ===============

FPCRType StandardFPSCRValue()
bits(32) upper = '00000000000000000000000000000000';
bits(32) lower = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '00000000000000000000';
return upper : lower;

Library pseudocode for aarch32/functions/memory-AArch32.CheckAlignment

// AArch32.CheckAlignment()
// =============

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype, boolean iswrite)
if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
    A = SCTLR[].A; //use AArch64 register, when higher Exception level is using AArch64
elsif PSTATE.EL == EL2 then
    A = HSCTLR.A;
else
    A = SCTLR.A;
aligned = (address == Align(address, alignment));
atomic = acctype IN { AccType_ATOMIC, AccType_ATOMICCRW, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICCRW, AccType_ATOMICLS64};
ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW, AccType_ATOMICLS64};
vector = acctype == AccType_VEC;
// AccType_VEC is used for SIMD element alignment checks only
check = (atomic || ordered || vector || A == '1');
if check && !aligned then
    secondstage = FALSE;
    AArch32.Abort(address, AArch32.AlignmentFault(acctype, iswrite, secondstage));
return aligned;
Library pseudocode for aarch32/functions/memory/AArch32.MemSingle

// AArch32.MemSingle[] - non-assignment (read) form
// ________________________________________________________________
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned]
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
bisize(size*8) value;
iswrite = FALSE;
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch32.Abort(address, memaddrdesc.fault);

// Memory array access
if HaveTME() then
  transactional = TSTATE.depth > 0 && !(acctype IN {AccType_IFETCH, AccType_TTW});
  accdesc = CreateAccessDescriptor(acctype, transactional);
else
  accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
  if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
    bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
    if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
      AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);
  value = _Mem[memaddrdesc, size, accdesc, FALSE];
return value;

// AArch32.MemSingle[] - assignment (write) form
// ________________________________________________________________
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) value
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;
memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
  AArch32.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareable then
  ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
if HaveTME() then
  transactional = TSTATE.depth > 0;
  accdesc = CreateAccessDescriptor(acctype, transactional);
else
  accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
  if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
    bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
    if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
      AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);
  _Mem[memaddrdesc, size, accdesc] = value;
return;
Library pseudocode for aarch32/functions/memory/Hint_PreloadData

Hint_PreloadData(bits(32) address);

Library pseudocode for aarch32/functions/memory/Hint_PreloadDataForWrite

Hint_PreloadDataForWrite(bits(32) address);

Library pseudocode for aarch32/functions/memory/Hint_PreloadInstr

Hint_PreloadInstr(bits(32) address);

Library pseudocode for aarch32/functions/memory/MemA

// MemA[] - non-assignment form
// ---------------------------------

bits(8*size) MemA[bits(32) address, integer size]
    acctype = AccType_ATOMIC;
    return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
// ---------------------------------

MemA[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ATOMIC;
    Mem_with_type[address, size, acctype] = value;
    return;

Library pseudocode for aarch32/functions/memory/MemO

// MemO[] - non-assignment form
// ---------------------------------

bits(8*size) MemO[bits(32) address, integer size]
    acctype = AccType_ORDERED;
    return Mem_with_type[address, size, acctype];

// MemO[] - assignment form
// ---------------------------------

MemO[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ORDERED;
    Mem_with_type[address, size, acctype] = value;
    return;

Library pseudocode for aarch32/functions/memory/MemU

// MemU[] - non-assignment form
// ---------------------------------

bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType_NORMAL;
    return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// ---------------------------------

MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_NORMAL;
    Mem_with_type[address, size, acctype] = value;
    return;
Library pseudocode for aarch32/functions/memory/MemU_unpriv

// MemU_unpriv[] - non-assignment form
// ===================================
bits(8*size) MemU_unpriv[bits(32) address, integer size]
   acctype = AccType_UNPRIV;
   return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// =================================
MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
   acctype = AccType_UNPRIV;
   Mem_with_type[address, size, acctype] = value;
   return;
Library pseudocode for aarch32/functions/memory/Mem_with_type

// Mem_with_type[] - non-assignment (read) form
// ===========================================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    boolean iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
    if !aligned then
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned];

        if BigEndian(acctype) then
            value = BigEndianReverse(value);

        return value;

// Mem_with_type[] - assignment (write) form
// ===========================================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value
    boolean iswrite = TRUE;

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);
    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch32.MemSingle[address, size, acctype, aligned] = value;

    return;
// AArch32.ESBOperation()
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

// Check if routed to AArch64 state
route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
  route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
  route_to_aarch64 = SCR_EL3.EA == '1';

if route_to_aarch64 then
  AArch64.ESBOperation();
  return;

route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.EA == '1';
route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE == '1' || HCR.AMO == '1');
if route_to_monitor then
  target = M32_Monitor;
elsif route_to_hyp || PSTATE.M == M32_Hyp then
target = M32_Hyp;
else
  target = M32_Abort;
endif

if IsSecure() then
  mask_active = TRUE;
elsif target == M32_Monitor then
  mask_active = SCR.AW == '1' && (!HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0'));
else
  mask_active = target == M32_Abort || PSTATE.M == M32_Hyp;
endif

mask_set = PSTATE.A == '1';
(-, el) = ELFromM32(target);
intdis = Halted() || ExternalDebugInterruptsDisabled(el);
masked = intdis || (mask_active && mask_set);

Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();
Library pseudocode for aarch32/functions/ras/AArch32.ReportDeferredSError

// AArch32.ReportDeferredSError()
// ================
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)
bits(32) target;
target<31> = '1'; // A
syndrome = Zeros(16);
if PSTATE.EL == EL2 then
    syndrome<11:10> = AET; // AET
    syndrome<9> = ExT; // EA
    syndrome<5:0> = '010001'; // DFSC
else
    syndrome<15:14> = AET; // AET
    syndrome<12> = ExT; // ExT
    syndrome<9> = TTBCR.EAE; // LPAE
    if TTBCR.EAE == '1' then // Long-descriptor format
        syndrome<5:0> = '010001'; // STATUS
    else // Short-descriptor format
        syndrome<10,3:0> = '10110'; // FS
if HaveAnyAArch64() then
    target<24:0> = ZeroExtend(syndrome); // Any RES0 fields must be set to zero
else
    target<15:0> = syndrome;
return target;

Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)

Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

// AArch32.vESBOperation()
// ==============
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
if 'ELUsingAArch32(EL2) then
    AArch64.vESBOperation();
return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
    // virtual SError interrupt might be pending
vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
vSEI_pending = vSEI_enabled && HCR.VA == '1';
vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
vmasked = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
if vSEI pending && vmasked then
    VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
    HCR.VA = '0'; // Clear pending virtual SError
return;
Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```c
// AArch32.ResetGeneralRegisters()
// ===============================
AArch32.ResetGeneralRegisters()
    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        Rmode[13, M32_Hyp] = bits(32) UNKNOWN;  // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        Rmode[i, M32_Monitor] = bits(32) UNKNOWN;
    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```c
// AArch32.ResetSIMDFPRegisters()
// ==============================
AArch32.ResetSIMDFPRegisters()
    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;
    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```c
// AArch32.ResetSpecialRegisters()
// ===============================
AArch32.ResetSpecialRegisters()
    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;
    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```c
AArch32.ResetSystemRegisters(boolean cold_reset);
```
Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// ================

ALUExceptionReturn(bits(32) address)
if PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.M IN {M32_User, M32_System} then
    Constraint c = ConstrainsUnpredictable(Unpredictable_ALUEXCEPTIONRETURN);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
case c of
    when Constraint_UNDEF
        UNDEFINED;
    when Constraint_NOP
        EndOfInstruction();
else
    AArch32.ExceptionReturn(address, SPSR[]);
```

Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
if CurrentInstrSet() == InstrSet_A32 then
    BXWritePC(address, BranchType_INDIR);
else
    BranchWritePC(address, BranchType_INDIR);
```

Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address, BranchType branch_type)
if address<0> == '1' then
    SelectInstrSet(InstrSet_T32);
    address<0> = '0';
else
    SelectInstrSet(InstrSet_A32);
    // For branches to an unaligned PC counter in A32 state, the processor takes the branch
    // and does one of:
    // * Forces the address to be aligned
    // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
    if address<1> == '1' && ConstrainsUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
        address<1> = '0';
    BranchTo(address, branch_type);
```

Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address, BranchType branch_type)
if CurrentInstrSet() == InstrSet_A32 then
    address<1:0> = '00';
else
    address<0> = '0';
    BranchTo(address, branch_type);
```
Library pseudocode for aarch32/functions/registers/D

// D[] - non-assignment form
// ===========================

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    return vreg<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    vreg<base+63:base> = value;
    V[n DIV 2] = vreg;
    return;

Library pseudocode for aarch32/functions/registers/Din

// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];

Library pseudocode for aarch32/functions/registers/LR

// LR - assignment form
// ====================

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// ========================

bits(32) LR
    return R[14];

Library pseudocode for aarch32/functions/registers/LoadWritePC

// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType_INDIR);
Library pseudocode for aarch32/functions/registers/LookUpRIndex

// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
   assert n >= 0 & & n <= 14;
   case n of // Select index by mode:     usr fiq irq svc abt und hyp
      when 8     result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
      when 9     result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
      when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
      when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
      when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
      when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
      when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
      otherwise  result = n;
   return result;

Library pseudocode for aarch32/functions/registers/Monitor_mode_registers

bits(32) SP_mon;
bits(32) LR_mon;

Library pseudocode for aarch32/functions/registers/PC

// PC - non-assignment form
// ==============

bits(32) PC
   return R[15];               // This includes the offset from AArch32 state

Library pseudocode for aarch32/functions/registers/PCStoreValue

// PCStoreValue()
// ==============

bits(32) PCStoreValue()
   // This function returns the PC value. On architecture versions before Armv7, it
   // is permitted to instead return PC+4, provided it does so consistently. It is
   // used only to describe A32 instructions, so it returns the address of the current
   // instruction plus 8 (normally) or 12 (when the alternative is permitted).
   return PC;

Library pseudocode for aarch32/functions/registers/Q

// Q[] - non-assignment form
// ==============

bits(128) Q(integer n)
   assert n >= 0 & & n <= 15;
   return V[n];

// Q[] - assignment form
// ==============

Q(integer n) = bits(128) value
   assert n >= 0 & & n <= 15;
   V[n] = value;
   return;
Library pseudocode for aarch32/functions/registers/Qin

// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];

Library pseudocode for aarch32/functions/registers/R

// R[] - assignment form
// ==============

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =========

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];

Library pseudocode for aarch32/functions/registers/RBankSelect

// RBankSelect()
// ===============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq, integer svc, integer abt, integer hyp)

    case mode of
        when M32_User result = usr; // User mode
        when M32_FIQ result = fiq; // FIQ mode
        when M32_IRQ result = irq; // IRQ mode
        when M32_Svc result = svc; // Supervisor mode
        when M32_Abort result = abt; // Abort mode
        when M32_Hyp result = hyp; // Hyp mode
        when M32_Undef result = und; // Undefined mode
        when M32_System result = usr; // System mode uses User mode registers
        otherwise Unreachable(); // Monitor mode

    return result;
Library pseudocode for aarch32/functionsregisters/Rmode

// Rmode[] - non-assignment form
// -------------------------------------------

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 & n <= 14;
    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);
    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;
// Rmode[] - assignment form
// -------------------------------------------

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 & n <= 14;
    // Check for attempted use of Monitor mode in Non-secure state.
    if !IsSecure() then assert mode != M32_Monitor;
    assert !BadMode(mode);
    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if !HighestELUsingAArch32() && ConstranUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;
        return;

Library pseudocode for aarch32/functionsregisters/S

// S[] - non-assignment form
// -------------------------------------------

bits(32) S[integer n]
    assert n >= 0 & n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    return vreg<base+31:base>;
// S[] - assignment form
// -------------------------------------------

S[integer n] = bits(32) value
    assert n >= 0 & n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    vreg<base+31:base> = value;
    V[n DIV 4] = vreg;
    return;
Library pseudocode for aarch32/functions/registers/SP

// SP - assignment form
// ====================

SP = bits(32) value
  R[13] = value;
  return;

// SP - non-assignment form
// =========================

bits(32) SP
  return R[13];

Library pseudocode for aarch32/functions/registers/_Dclone

array bits(64) _Dclone[0..31];

Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

// AArch32.ExceptionReturn()
// =========================

AArch32.ExceptionReturn(bits(32) new_pc, bits(32) spsr)
  SynchronizeContext();
  // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
  // mechanism
  SetPSTATEFromPSR(spsr);
  ClearExclusiveLocal(ProcessorID());
  SendEventLocal();
  if PSTATE.IL == '1' then
    // If the exception return is illegal, PC[1:0] are UNKNOWN
    new_pc<1:0> = bits(2) UNKNOWN;
  else
    // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
    if PSTATE.T == '1' then
      new_pc<0> = '0';                 // T32
    else
      new_pc<1:0> = '00';              // A32
    BranchTo(new_pc, BranchType_ERET);

Library pseudocode for aarch32/functions/system/AArch32ExecutingATS1xPInstr

// AArch32ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1CPR/WP

boolean AArch32ExecutingATS1xPInstr()
  if !HavePrivATExt() then return FALSE;

  instr = ThisInstr();
  if instr<24+:4> == '1110' && instr<8+:4> == '1111' then
    opc1 = instr<21+:3>;
    CRn = instr<16+:4>;
    CRm = instr<0+:4>;
    opc2 = instr<5+:3>;
    return (opc1 == '000' && CRn == '0111' && CRm == '1001' && opc2 IN {'000','001'});
  else
    return FALSE;
Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

// AArch32.ExecutingCP10or11Instr()
// ===============================================

boolean AArch32.ExecutingCP10or11Instr()
    instr = ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN (InstrSet_A32, InstrSet_T32);
    if instr_set == InstrSet_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');
    else // InstrSet_T32
        return (instr<31:28> == '111x' && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> == '101x');

Library pseudocode for aarch32/functions/system/AArch32.ExecutingLSMInstr

// AArch32.ExecutingLSMInstr()
// ===========================
// Returns TRUE if processor is executing a Load/Store Multiple instruction

boolean AArch32.ExecutingLSMInstr()
    instr = ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN (InstrSet_A32, InstrSet_T32);
    if instr_set == InstrSet_A32 then
        return (instr<28+:4> != '1111' && instr<25+:3> == '100');
    else // InstrSet_T32
        if ThisInstrLength() == 16 then
            return (instr<12+:4> == '1100');
        else
            return (instr<25+:7> == '1110100' && instr<22> == '0');

Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

// Read from a 32-bit AArch32 System register and return the register's contents.
bits(32) AArch32.SysRegRead(integer cp_num, bits(32) instr);

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

// Read from a 64-bit AArch32 System register and return the register's contents.
bits(64) AArch32.SysRegRead64(integer cp_num, bits(32) instr);
boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14, 15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;
    return FALSE;

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

// Write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

// Write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);

Library pseudocode for aarch32/functions/system/AArch32.WriteMode

// AArch32.WriteMode()
// ===============
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.
AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M = mode;
    PSTATE.EL = el;
    PSTATE.nRW = '1';
    PSTATE.SP = (if mode IN {M32_User, M32_System} then '0' else '1');
    return;
// AArch32.WriteModeByInstr()
// =========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode) {
  (valid, el) = ELFromM32(mode);
  // 'valid' is set to FALSE if 'mode' is invalid for this implementation or the current value
  // of SCR.NS/SCR.EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
  // PSTATE.EL if it would result in any of:
  // * A change to a mode that would cause entry to a higher Exception level.
  if UInt(el) > UInt(PSTATE.EL) then
    valid = FALSE;
  // * A change to or from Hyp mode.
  if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
    valid = FALSE;
  // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
  if PSTATE.M == M32_Monitor && HaveAArch32EL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
    valid = FALSE;
  if !valid then
    PSTATE.IL = '1';
  else
    AArch32.WriteMode(mode);
}

// BadMode()
// =========

boolean BadMode(bits(5) mode) {
  // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
  case mode of
    when M32_Monitor
      valid = HaveAArch32EL(EL3);
    when M32_Hyp
      valid = HaveAArch32EL(EL2);
    when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
      // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
      // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
      // AArch64, then these modes are EL1 modes.
      // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
      valid = HaveAArch32EL(EL1);
    when M32_User
      valid = HaveAArch32EL(EL0);
    otherwise
      valid = FALSE;  // Passed an illegal mode value
  return !valid;
}
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100'  // R8_usr to R12_usr
            if mode != M32_FIQ then UNPREDICTABLE;
        when '00101'               // SP_usr
            if mode == M32_System then UNPREDICTABLE;
        when '00110'               // LR_usr
            if mode IN {M32_Hyp,M32_System} then UNPREDICTABLE;
        when '010xx', '0110x', '01110'  // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '1000x'               // LR_irq, SP_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '1001x'  // LR_svc, SP_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '1010x'               // LR_abt, SP_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '1011x'               // LR_und, SP_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '1110x'               // LR_mon, SP_mon
            if !HaveEL(EL3) || !IsSecure() || mode == M32_Monitor then UNPREDICTABLE;
        when '11110'               // ELR_hyp, only from Monitor or Hyp mode
            if !HaveEL(EL2) || !mode IN {M32_Monitor,M32_Hyp} then UNPREDICTABLE;
        when '11111'               // SP_hyp, only from Monitor mode
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```c
// CPSRWriteByInstr()
// ===============
CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;          // PSTATE.<A,I,F,M> are not writable at EL0
    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>; // Bits <26:24> are ignored
    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;
    if bytemask<1> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.E = value<9>;                    // PSTATE.E is writable at EL0
    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>; // Bit <5> is RES0
        // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
        AArch32.WriteModeByInstr(value<4:0>);
    return;
```

Library pseudocode for aarch32/functions/system/ConditionPassed

```c
// ConditionPassed()
// ===============

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

Library pseudocode for aarch32/functions/system/CurrentCond

```c
bits(4) AArch32.CurrentCond();
```

Library pseudocode for aarch32/functions/system/InITBlock

```c
// InITBlock()
// ===========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```
library pseudocode for aarch32/functions/system/LastInITBlock

// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');

library pseudocode for aarch32/functions/system/SPSRWriteByInstr

// SPSRWriteByInstr()
// ================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    bits(32) new_spsr = SPSR[];
    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>; // N,Z,C,V,Q flags, IT[1:0],J bits
    if bytemask<2> == '1' then
    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>; // IT[7:2] bits, E bit, A interrupt mask
    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>; // I,F interrupt masks, T bit, Mode bits
    SPSR[] = new_spsr; // UNPREDICTABLE if User or System mode
    return;

library pseudocode for aarch32/functions/system/SPSRaccessValid

// SPSRaccessValid()
// ===============

// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
      when '01110' // SPSR_fiq
        if mode == M32_FIQ then UNPREDICTABLE;
      when '10000' // SPSR_irq
        if mode == M32_IRQ then UNPREDICTABLE;
      when '10010' // SPSR_svc
        if mode == M32_Svc then UNPREDICTABLE;
      when '10100' // SPSR_abt
        if mode == M32_Abort then UNPREDICTABLE;
      when '10110' // SPSR_und
        if mode == M32_Undef then UNPREDICTABLE;
      when '11100' // SPSR_mon
        if !HaveEL(EL3) || mode == M32_Monitor || !IsSecure() then UNPREDICTABLE;
      when '11110' // SPSR_hyp
        if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
      otherwise
        UNPREDICTABLE;
    return;
Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
  assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
  assert iset IN {InstrSet_A32, InstrSet_T32};
  PSTATE.T = if iset == InstrSet_A32 then '0' else '1';
  return;
```

Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
  result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
  return result;
```

Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
  (result, -) = SignedSatQ(i, N);
  return result;
```

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
  (result, -) = UnsignedSatQ(i, N);
  return result;
```
// AArch32.CombineS1S2Desc()
// =========================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor AArch32.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc, AccType s2acctype)

AddressDescriptor result;
result.paddress = s2desc.paddress;

apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
if IsFault(s1desc) || IsFault(s2desc) then
    result = if IsFault(s1desc) then s1desc else s2desc;
else
    result.fault = AArch32.NoFault();
    if s2desc.memattrs.memtype == MemType_Device || (!apply_force_writeback && s1desc.memattrs.memtype == MemType_Device) then
        result.memattrs.memtype = MemType_Device;
    elseif s2desc.memattrs.memtype == MemType_Normal then
        result.memattrs.device = s2desc.memattrs.device;
    else // Both Device
        result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                   s2desc.memattrs.device);
    result.memattrs.tagged = FALSE;
    // S1 can be either Normal or Device, S2 is Normal.
    else
        result.memattrs.memtype = MemType_Normal;
        result.memattrs.device = DeviceType UNKNOWN;
        result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner, s2acctype);
        result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer, s2acctype);
        result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
        result.memattrsoutershareable = (s1desc.memattrsoutershareable || s2desc.memattrsoutershareable);
        result.memattrs.tagged = (s1desc.memattrs.tagged &&
                                 result.memattrs.inner.attrs == MemAttr_WB &&
                                 result.memattrs.inner.hints == MemHint_RWA &&
                                 result.memattrs.outer.attrs == MemAttr_WB &&
                                 result.memattrs.outer.hints == MemHint_RWA);

result.memattrs = MemAttrDefaults(result.memattrs);
return result;
MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB);
        TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;
    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRnE;
        when '00001', '01000'
            // Device-nGnRE
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRE;
        when '00010', '00011', '00100'
            // Write-back or Write-through Read allocate, or Non-cacheable
            memattrs.memtype = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints(C:B, acctype, FALSE);
            memattrs.shareable = (S == '1');
        when '00110'
            memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            memattrs.memtype = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints('01', acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints('01', acctype, FALSE);
            memattrs.shareable = (S == '1');
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            memattrs.memtype = MemType_Normal;
            memattrs.inner = ShortConvertAttrsHints(C:B, acctype, FALSE);
            memattrs.outer = ShortConvertAttrsHints(TEX<1:0>, acctype, FALSE);
            memattrs.shareable = (S == '1');
        otherwise
            // Reserved, handled above
            Unreachable();
    endcase;

    // transient bits are not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;

    // distinction between inner and outer shareable is not supported in this format
    memattrs.outershareable = memattrs.shareable;
    memattrs.tagged = FALSE;

    return MemAttrDefaults(memattrs);
AArch32.InstructionDevice()
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch32.InstructionDevice(AddressDescriptor addrdesc, bits(32) vaddress, bits(40) ipaddress, integer level, bits(4) domain, AccType accType, boolean iswrite, boolean secondstage, boolean s2fs1walk)

c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
assert c IN {Constraint_NONE, Constraint_FAULT};

if c == Constraint_FAULT then
    addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, level, accType, iswrite, secondstage, s2fs1walk);
else
    addrdesc.memattrs.memtype = MemType_Normal;
    addrdesc.memattrs.inner.attrs = MemAttr_NC;
    addrdesc.memattrs.inner.hints = MemHint_No;
    addrdesc.memattrs.outer = addrdesc.memattrs.inner;
    addrdesc.memattrs.tagged = FALSE;
    addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

return addrdesc;
// AArch32.RemappedTEXDecode()
// ===========================

MemoryAttributes AArch32.RemappedTEXDecode(int TEX, bit C, bit B, bit S, AccType acctype)

    MemoryAttributes memattrs;

    region = UInt(TEX<0>:C:B);         // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;
    else
        base = 2 * region;
        attrfield = PRRR<base+1:base>;
        if attrfield == '11' then      // Reserved, maps to allocated value
            (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESPRR);
            case attrfield of
                when '00'                  // Device-nGnRnE
                    memattrs.memtype = MemType_Device;
                    memattrs.device = DeviceType_nGnRnE;
                when '01'                  // Device-nGnRE
                    memattrs.memtype = MemType_Device;
                    memattrs.device = DeviceType_nGnRE;
                when '10'
                    memattrs.memtype = MemType_Normal;
                    memattrs.inner = ShortConvertAttrsHints(NMRR<base+1:base>, acctype, FALSE);
                    memattrs.outer = ShortConvertAttrsHints(NMRR<base+17:base+16>, acctype, FALSE);
                    s_bit = if S == '0' then PRRR.NS0 else PRRR.NS1;
                    memattrs.shareable = (s_bit == '1');
                    memattrs.outershareable = (s_bit == '1' && PRRR<region+24> == '0');
                when '11'
                    Unreachable();
        endcase
        memattrs.inner.transient = FALSE;
        memattrs.outer.transient = FALSE;
        memattrs.tagged = FALSE;
    endif
    return MemAttrDefaults(memattrs);
// AArch32.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch32.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;
    if PSTATE.EL == EL2 then
        mair = HMAIR1:HMAIR0;
    else
        mair = MAIR1:MAIR0;
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    memattrs.tagged = FALSE;
    if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
    if !HaveMTEExt() && attrfield<7:4> == '1111' && attrfield<3:0> == '0000' then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

    if attrfield<7:4> == '0000' then            // Device
        memattrs.memtype = MemType_Device;
        case attrfield<3:0> of
            when '0000' memattrs.device = DeviceType_nGnRnE;
            when '0100' memattrs.device = DeviceType_nGnRE;
            when '1000' memattrs.device = DeviceType_nGRE;
            when '1100' memattrs.device = DeviceType_GRE;
            otherwise Unreachable();         // Reserved, handled above
        elsif attrfield<3:0> != '0000' then        // Normal
            memattrs.memtype = MemType_Normal;
            memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
            memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
            memattrs.shareable = SH<1> == '1';
            memattrsoutershareable = SH == '10';
        elsif HaveMTEExt() && attrfield == '11100000' then // Normal, Tagged WB-RWA
            memattrs.memtype = MemType_Normal;
            memattrs.outer = LongConvertAttrsHints('1111', acctype); // WB_RWA
            memattrs.inner = LongConvertAttrsHints('1111', acctype); // WB_RWA
            memattrs.shareable = SH<1> == '1';
            memattrsoutershareable = SH == '10';
            memattrs.tagged = TRUE;
        else
            Unreachable();                          // Reserved, handled above
    return MemAttrDefaults(memattrs);
Library pseudocode for aarch32/translation/attrs/AArch32.TranslateAddressS1Off

// AArch32.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch32.TranslateAddressS1Off(bits(32) vaddress, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());

TLBRecord result;
result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;

default_cacheable = (HasS2Translation() && ((if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC) == '1'));

if default_cacheable then
    // Use default cacheable settings
    result.addrdesc.memattrs.memtype = MemType_Normal;
    result.addrdesc.memattrs.inner.attrs = MemAttr_WB;       // Write-back
    result.addrdesc.memattrs.inner.hints = MemHint_RWA;
    result.addrdesc.memattrs.shareable = FALSE;
    result.addrdesc.memattrs.outershareable = FALSE;
    result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
elsif acctype != AccType_IFETCH then
    // Treat data as Device
    result.addrdesc.memattrs.memtype = MemType_Device;
    result.addrdesc.memattrs.device = DeviceType_nGnRnE;
    result.addrdesc.memattrs.inner = MemAttrHints_UNKNOWN;
    result.addrdesc.memattrs.tagged = FALSE;
else
    // Instruction cacheability controlled by SCTLR/HSCTLR.I
    if PSTATE.EL == EL2 then
        cacheable = HSCTLR.I == '1';
    else
        cacheable = SCTLR.I == '1';
    result.addrdesc.memattrs.memtype = MemType_Normal;
    if cacheable then
        result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
        result.addrdesc.memattrs.inner.hints = MemHint_RA;
    else
        result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
        result.addrdesc.memattrs.inner.hints = MemHint_No;
    result.addrdesc.memattrs.shareable = TRUE;
    result.addrdesc.memattrs.outershareable = TRUE;
    result.addrdesc.memattrs.tagged = FALSE;
result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;
result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

result.perms.ap = bits(3) UNKNOWN;
result.perms.xn = '0';
result.perms.pxn = '0';

result.nG = bit UNKNOWN;
result.contiguous = boolean UNKNOWN;
result.domain = bits(4) UNKNOWN;
result.level = integer UNKNOWN;
result.blocksize = integer UNKNOWN;
result.addrdesc.paddress.address = ZeroExtend(vaddress);
result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
result.addrdesc.fault = AArch32.NoFault();

result.descupdate.descaddr = result.addrdesc;
return result;
// AArch32.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.
bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;

// AArch32.CheckDomain()
// =====================
(boolean, FaultRecord) AArch32.CheckDomain(bits(4) domain, bits(32) vaddress, integer level, AccType acctype, boolean iswrite)
    index = 2 * UInt(domain);
    attrfield = DACR<index+1:index>;
    if attrfield == '10' then          // Reserved, maps to an allocated value
       #ga progn 32768
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESDACR);
    if attrfield == '00' then
        fault = AArch32.DomainFault(domain, level, acctype, iswrite);
    else
        fault = AArch32.NoFault();
    permissioncheck = (attrfield == '01');
    return (permissioncheck, fault);
Library pseudocode for aarch32/translation/checks/AArch32.CheckPermission
// AArch32.CheckPermission()
// Function used for permission checking from AArch32 stage 1 translations

FaultRecord AArch32.CheckPermission(Permissions perms, bits(32) vaddress, integer level, bits(4) domain, bit NS, AccType acctype, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());

if PSTATE.EL != EL2 then
    wxn = SCTLR.WXN == '1';
    if TTBCR.EAE == '1' || SCTLR.AFE == '1' || perms.ap<0> == '1' then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';
    else
        priv_r = perms.ap<2:1> != '00';
        priv_w = perms.ap<2:1> == '01';
        user_r = perms.ap<1> == '1';
        user_w = FALSE;
    uwxn = SCTLR.UWXN == '1';
    ispriv = AArch32.AccessUsesEL(acctype) != EL0;

    user_xn = !user_r || perms.xn == '1' || (user_w && wxn);
    priv_xn = (!priv_r || perms.xn == '1' || perms.pxn == '1' ||
        (priv_w && wxn) || (user_w && uwxn));
    pan = if HavePANExt() then PSTATE.PAN else '0';
    is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
    is_ats1xp = (acctype == AccType_AT && AArch32.ExecutingATS1XPInstr());
    if pan == '1' && user_r && ispriv && (is_ldst || is_ats1xp) then
        priv_r = FALSE;
        priv_w = FALSE;
    else
        // Access from EL2
        wxn = HSCTLR.WXN == '1';
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' then
        secure_instr_fetch = if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF;
        if secure_instr_fetch == '1' then xn = TRUE;
    else
        if acctype == AccType_IFETCH then
            fail = xn;
            failedread = TRUE;
        elsif acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDDATOMICRW} then
            fail = !r || !w;
            failedread = !r;
        elsif acctype == AccType_DC then
            // DC maintenance instructions operating by VA, cannot fault from stage 1 translation.
            fail = FALSE;
        elsif iswrite then
            fail = !w;
            failedread = FALSE;
        else
            fail = !r;
            failedread = TRUE;
    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
Library pseudocode for aarch32/translation/checks/AArch32.CheckS2Permission

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress, integer level, AccType acctype, boolean iswrite, boolean s2fs1walk)

assert HaveEL(EL2) && !IsSecure() && EUsingAArch32(EL2) && HasS2Translation();

r = perms.ap<1> == '1';
w = perms.ap<2> == '1';
if HaveExtendedExecuteNeverExt() then
  case perms.xn:perms.xxn of
    when '00'  xn = !r;
    when '01'  xn = !r || PSTATE.EL == EL1;
    when '10'  xn = TRUE;
    when '11'  xn = !r || PSTATE.EL == EL0;
  else
    xn = !r || perms.xn == '1';

  // Stage 1 walk is checked as a read, regardless of the original type
  if acctype == AccType_IFETCH && !s2fs1walk then
    fail = xn;
    failedread = TRUE;
  elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW }) && !s2fs1walk then
    fail = !r || !w;
    failedread = !r;
  elsif acctype == AccType_DC && !s2fs1walk then
    // DC maintenance instructions operating by VA, do not generate Permission faults
    // from stage 2 translation, other than from stage 1 translation table walk.
    fail = FALSE;
  elsif iswrite && !s2fs1walk then
    fail = !w;
    failedread = FALSE;
  else
    fail = !r;
    failedread = !iswrite;

  if fail then
    domain = bits(4) UNKNOWN;
    secondstage = TRUE;
    return AArch32.PermissionFault(ipaddress, domain, level, acctype, !failedread, secondstage, s2fs1walk);
  else
    return AArch32.NoFault();

Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```
AArch32.CheckBreakpoint()
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());
assert size IN {2,4};

match = FALSE;
mismatch = FALSE;
for i = 0 to UInt(DBGDIDR.BRPs)
  (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
  match = match || match_i;
  mismatch = mismatch || mismatch_i;
if match && HaltOnBreakpointOrWatchpoint() then
  reason = DebugHalt_Breakpoint;
  Halt(reason);
elsif (match || mismatch) then
  acctype = AccType_IFETCH;
  iswrite = FALSE;
  debugmoe = DebugException_Breakpoint;
  return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
  return AArch32.NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```
AArch32.CheckDebug()
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

FaultRecord fault = AArch32.NoFault();
d_side = (acctype != AccType_IFETCH);
generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
halt = HaltOnBreakpointOrWatchpoint();
// Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);
if !d_side && vector_catch_first && generate_exception then
  fault = AArch32.CheckVectorCatch(vaddress, size);
if fault.statuscode == Fault_None && (generate_exception || halt) then
  if d_side then
    fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
  else
    fault = AArch32.CheckBreakpoint(vaddress, size);
if fault.statuscode == Fault_None && !d_side && !vector_catch_first && generate_exception then
  return AArch32.CheckVectorCatch(vaddress, size);
return fault;
```
Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

// AArch32.CheckVectorCatch()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());

match = AArch32.VCMatch(vaddress);
if size == 4 && !match && AArch32.VCMatch(vaddress + 2) then
    match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

if match then
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    debugmoe = DebugException_VectorCatch;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();

Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)
assert ELUsingAArch32(S1TranslationRegime());

if acctype IN {AccType_TTW, AccType_IC, AccType_AT} then
    return AArch32.NoFault();
if acctype == AccType_DC then
    if !iswrite then
        return AArch32.NoFault();
    elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
        return AArch32.NoFault();

match = FALSE;
ispriv = AArch32.AccessUsesEL(acctype) != EL0;

for i = 0 to UInt(DBGDIDR.WRPs)
    match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite);

if match && HaltonBreakpointOrWatchpoint() then
    reason = DebugHalt_Watchpoint;
    EDWAR = vaddress;
    Halt(reason);
elsif match then
    debugmoe = DebugException_Watchpoint;
    return AArch32.DebugFault(acctype, iswrite, debugmoe);
else
    return AArch32.NoFault();
Library pseudocode for aarch32/translation/faults/AArch32.AccessFlagFault

// AArch32.AccessFlagFault()
// =========================

FaultRecord AArch32.AccessFlagFault(bits(40) ipaddress, bits(4) domain, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)

extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
erortype = bits(2) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_AccessFlag, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch32/translation/faults/AArch32.AddressSizeFault

// AArch32.AddressSizeFault()
// ===========================

FaultRecord AArch32.AddressSizeFault(bits(40) ipaddress, bits(4) domain, integer level, AccType acctype, boolean iswrite, boolean secondstage, boolean s2fs1walk)

extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
erortype = bits(2) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_AddressSize, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch32/translation/faults/AArch32.AlignmentFault

// AArch32.AlignmentFault()
// ========================

FaultRecord AArch32.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

ipaddress = bits(40) UNKNOWN;
domain = bits(4) UNKNOWN;
level = integer UNKNOWN;
extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
erortype = bits(2) UNKNOWN;
s2fs1walk = boolean UNKNOWN;

return AArch32.CreateFaultRecord(Fault_Alignment, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch32/translation/faults/AArch32.AsynchExternalAbort

// AArch32.AsynchExternalAbort()
// =============================

// Wrapper function for asynchronous external aborts

FaultRecord AArch32.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

faulttype = if parity then Fault_AsyncParity else Fault_AsyncExternal;
ipaddress = bits(40) UNKNOWN;
domain = bits(4) UNKNOWN;
level = integer UNKNOWN;
extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
acctype = AccType_NORMAL;
iswrite = boolean UNKNOWN;
debugmoe = bits(4) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;

return AArch32.CreateFaultRecord(faulttype, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, errortype, secondstage, s2fs1walk);
Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

// AArch32.DebugFault()
// ===============

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Debug, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch32/translation/faults/AArch32.DomainFault

// AArch32.DomainFault()
// ================

FaultRecord AArch32.DomainFault(bits(4) domain, integer level, AccType acctype, boolean iswrite)

    ipaddress = bits(40) UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_Domain, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch32/translation/faults/AArch32.NoFault

// AArch32.NoFault()
// ===============

FaultRecord AArch32.NoFault()

    ipaddress = bits(40) UNKNOWN;
    domain = bits(4) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    debugmoe = bits(4) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch32.CreateFaultRecord(Fault_None, ipaddress, domain, level, acctype, iswrite, extflag, debugmoe, errortype, secondstage, s2fs1walk);
Library pseudocode for aarch32/translation/faults/AArch32.PermissionFault

// AArch32.PermissionFault()
// =========================

FaultRecord AArch32.PermissionFault(bits(40) ipaddress, bits(4) domain, integer level, 
AccType acctype, boolean iswrite, boolean secondstage, 
boolean s2fs1walk)

extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
errortype = bits(2) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_Permission, ipaddress, domain, level, acctype, iswrite, 
extflag, debugmoe, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch32/translation/faults/AArch32.TranslationFault

// AArch32.TranslationFault()
// =========================

FaultRecord AArch32.TranslationFault(bits(40) ipaddress, bits(4) domain, integer level, 
AccType acctype, boolean iswrite, boolean secondstage, 
boolean s2fs1walk)

extflag = bit UNKNOWN;
debugmoe = bits(4) UNKNOWN;
errortype = bits(2) UNKNOWN;
return AArch32.CreateFaultRecord(Fault_Translation, ipaddress, domain, level, acctype, iswrite, 
extflag, debugmoe, errortype, secondstage, s2fs1walk);
// AArch32.FirstStageTranslate()
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch32.FirstStageTranslate(bits(32) vaddress, AccType accctype, boolean iswrite, boolean wasaligned, integer size)

if PSTATE.EL == EL2 then
    s1_enabled = HSCTLR.M == '1';
elsif EL2Enabled() then
    tge = (if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE);
    dc = (if ELUsingAArch32(EL2) then HCR.DC else HCR_EL2.DC);
    s1_enabled = tge == '0' && dc == '0' && SCTLR.M == '1';
else
    s1_enabled = SCTLR.M == '1';

TLBRecord S1;
S1.addrdesc.fault = AArch32.NoFault();
ipaddress = bits(40) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;

if s1_enabled then                             // First stage enabled
    use_long_descriptor_format = PSTATE.EL == EL2 || TTBCR.EAE == '1';
    if use_long_descriptor_format then
        S1 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, accctype, iswrite, secondstage, s2fs1walk, size);
        permissioncheck = TRUE;  domaincheck = FALSE;
    else
        S1 = AArch32.TranslationTableWalkSD(vaddress, accctype, iswrite, size);
        permissioncheck = TRUE;  domaincheck = TRUE;
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, accctype, iswrite);
        permissioncheck = FALSE;  domaincheck = FALSE;
InGuardedPage = FALSE;                 // No memory is guarded when stage 1 address translation is

    if !IsFault(S1.addrdesc) && UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() &&
        S1.addrdesc.memattrs.memtype == MemType_Device && S1.addrdesc.memattrs.device != DeviceType_GRE then
        nTLSMD = if S1TranslationRegime() == EL2 then HSCTLR.nTLSMD else SCTLR.nTLSMD;
        if nTLSMD == '0' then
            S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
        // Check for unaligned data accesses to Device memory
        if (!wasaligned && acctype != AccType_IFETCH || acctype == AccType_DCZVA) &&
            !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device then
                S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
        if !Isfault(S1.addrdesc) && domaincheck && (!acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC}) then
            (permissioncheck, abort) = AArch32.CheckDomain(S1.domain, vaddress, S1.level, acctype, iswrite);
            S1.addrdesc.fault = abort;
        if !IsFault(S1.addrdesc) && permissioncheck then
            S1.addrdesc.fault = AArch32.CheckPermission(S1.perms, vaddress, S1.level, S1.domain, S1.addrdesc.paddress.NS, acctype, iswrite);
    // Check for instruction fetches from Device memory that has not been marked as execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch32.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level, S1.domain, acctype, iswrite, secondstage, s2fs1walk);
return S1.addrdesc;
// AArch32.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch32.FullTranslate(bits(32) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

// First Stage Translation
S1 = AArch32.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
if !IsFault(S1) && HasS2Translation() then
  s2fs1walk = FALSE;
  result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk, size);
else
  result = S1;
return result;
AddressDescriptor AArch32.SecondStageTranslate(AddressDescriptor s1, bits(32) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, boolean s2fs1walk, integer size)

assert HasS2Translation();
assert IsZero(s1.paddress.address<47:40>);
hwupdatewalk = FALSE;
if !ELUsingAArch32(EL2) then
  return AArch64.SecondStageTranslate(s1, ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned, s2fs1walk, size, hwupdatewalk);

s2_enabled = HCR.VM == '1' || HCR.DC == '1';
secondstage = TRUE;
if s2_enabled then // Second stage enabled
  ipaddress = s1.paddress.address<39:0>;
  S2 = AArch32.TranslationTableWalkLD(ipaddress, vaddress, acctype, iswrite, secondstage, s2fs1walk, size);

  // Check for unaligned data accesses to Device memory
  if (((!wasaligned && acctype != AccType_IFETCH) ||
       (acctype == AccType_DCZVA && !s2fs1walk)) &
     S2.addrdesc.memattrs.memtype == MemType_Device &
     !IsFault(S2.addrdesc)) then
    S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

  // Check for permissions on Stage2 translations
  if !IsFault(S2.addrdesc) then
    S2.addrdesc.fault = AArch32.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level, acctype, iswrite, s2fs1walk);

  // Check for instruction fetches from Device memory not marked as execute-never. As there
  // has not been a Permission Fault then the memory is not marked execute-never.
  if (!s2fs1walk &&
      !IsFault(S2.addrdesc) &&
      S2.addrdesc.memattrs.memtype == MemType_Device &&
      acctype == AccType_IFETCH) then
    domain = bits(4) UNKNOWN;
    S2.addrdesc = AArch32.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level, domain, acctype, iswrite, secondstage, s2fs1walk);

  if (s2fs1walk &&
      !IsFault(S2.addrdesc) &&
      S2.addrdesc.memattrs.memtype == MemType_Device) then
    // Check for protected table walk.
    if HCR.PTW == '1' then
      domain = bits(4) UNKNOWN;
      S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype, iswrite, secondstage, s2fs1walk);
    else
      // Translation table walk occurs as Normal Non-cacheable memory.
      S2.addrdesc.memattrs.memtype = MemType_Normal;
      S2.addrdesc.memattrs.inner.attrs = MemAttr_NC;
      S2.addrdesc.memattrs.outer.attrs = MemAttr_NC;
      S2.addrdesc.memattrs.shareable = TRUE;
      S2.addrdesc.memattrsoutershareable = TRUE;

    if s2fs1walk then
      result = AArch32.CombineS1S2Desc(S1, S2.addrdesc, AccType_TTW);
    else
      result = AArch32.CombineS1S2Desc(S1, S2.addrdesc, acctype);
  else
    result = S1;
return result;
// AArch32.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation table walk access.

AddressDescriptor AArch32.SecondStageWalk(AddressDescriptor S1, bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

assert HasS2Translation();

s2fs1walk = TRUE;
wasaligned = TRUE;
return AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk, size);

// AArch32.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

if !ELUsingAArch32(S1TranslationRegime()) then
    return AArch64.TranslateAddress(ZeroExtend(vaddress, 64), acctype, iswrite, wasaligned, size);
result = AArch32.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
if !IsFault(result) then
    result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);
// Update virtual address for abort functions
result.vaddress = ZeroExtend(vaddress);
return result;
AArch32.TranslationTableWalkLD()
-----------------------------
Returns a result of a translation table walk using the Long-descriptor format

Implementations might cache information from memory in any number of non-coherent TLB
 caching structures, and so avoid memory accesses that have been expressed in this
 pseudo code. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkLD(bits(40) ipaddress, bits(32) vaddress,  
 AccType acctype, boolean iswrite, boolean secondstage, 
 boolean s2fs1walk, integer size)

if !secondstage then
  assert ELUsingAArch32(S1TranslationRegime());
else
  assert HaveEL(EL2) && !IsSecure() && ELUsingAArch32(EL2) && HasS2Translation();

TLBRecord result;
AddressDescriptor descaddr;
bits(64) baseregister;
bits(40) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
bit nswalk;                    // Stage 2 translation table walks are to Secure or to Non-secure PA space

result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;
domain = bits(4) UNKNOWN;
descaddr.memattrs.memtype = MemType_Normal;

// Fixed parameters for the translation table walk:
//  grainsize = Log2(Size of Table)         - Size of Table is 4KB in AArch32
//  stride = Log2(Address per Level)        - Bits of address consumed at each level
constant integer grainsize = 12;                    // Log2(4KB page size)
constant integer stride = grainsize - 3;            // Log2(page size / 8 bytes)

// Derived parameters for the translation table walk:
//  inputsize = Log2(Size of Input Address) - Input Address size in bits
//  level = Level to start walk from
//  This means that the number of levels after start level = 3-level
if !secondstage then
  // First stage translation
  inputaddr = ZeroExtend(vaddress);
el = AArch32.AccessUsesEL(acctype);
isprivileged = AArch32.AccessUsesEL(acctype) != EL0;
if el == EL2 then
  inputsize = 32 - UInt(HTCR.T0SZ);
  basefound = inputsize == 32 || IsZero(inputaddr<31:inputsize>);
  disabled = FALSE;
  baseregister = HTTB0;
descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCR.ORGN0, HTCR.IRGN0, secondstage);
  reversedescriptors = HSCTLR.EE == '1';
  lookupsecure = FALSE;
  singlepriv = TRUE;
  hierattrssdisabled = AArch32.HaveHPDExt() && HTCR.HPD == '1';
else
  basefound = FALSE;
  disabled = FALSE;
t0size = UInt(TTBCR.T0SZ);
if t0size == 0 || IsZero(inputaddr<31:(32-t0size)>)) then
  inputsize = 32 - t0size;
  basefound = TRUE;
  baseregister = TTBR0;
descaddr.memattrs = WalkAttrDecode(TTBCR.SH0, TTBCR.ORGN0, TTBCR.IRGN0, secondstage);
  hierattrssdisabled = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD0 == '1';
tlsize = UInt(TTBT1SZ);
if (tlsize == 0 && !basefound) || (tlsize > 0 && IsOnes(inputaddr<31:(32-tlsize)>)) then
  inputsize = 32 - tlsize;
  basefound = TRUE;
  baseregister = TTBR1;
descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGN1, TTBCR.IGRN1, secondstage);
hierattrsdisabled = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD1 == '1';
reversedescriptors = SCTLR.EE == '1';
lookupsecure = IsSecure();
singlepriv = FALSE;

// The starting level is the number of strides needed to consume the input address
level = 4 - (1 + (inputsize - grainsize - 1) DIV stride));

else
// Second stage translation
inputaddr = ipaddress;
inputsize = 32 - SInt(VTCR.T0SZ);
// VTCR.S must match VTCR.T0SZ[3]
if VTCR.S != VTCR.T0SZ<3> then
  (-, inputsize) = ConstrainUnpredictableInteger(32-7, 32+8, Unpredictable_RESVTCRS);
  basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
disabled = FALSE;
descaddr.memattrs = WalkAttrDecode(VTCR.SH0, VTCR.ORGN0, VTCR.IGRN0, secondstage);
reversedescriptors = HSCTLR.EE == '1';
singlepriv = TRUE;
lookupsecure = FALSE;
baseregister = VTTBR;
startlevel = UInt(VTCR.SL0);
level = 2 - startlevel;
if level <= 0 then basefound = FALSE;

// Number of entries in the starting level table =
//   (Size of Input Address)(Address per level)^((Num levels remaining)*(Size of Table))
//   Log2(Num of entries)
//   Check for starting level table with fewer than 2 entries or longer than 16 pages.
//   Lower bound check is: startsizecheck < Log2(2 entries)
//   That is, VTCR.SL0 == '00' and SInt(VTCR.T0SZ) > 1, Size of Input Address < 2^31 bytes
//   Upper bound check is: startsizecheck > Log2(pagesize/8*16)
//   That is, VTCR.SL0 == '01' and SInt(VTCR.T0SZ) < -2, Size of Input Address > 2^34 bytes
// if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;
if !basefound || disabled then
  level = 1; // AArch64 reports this as a level 0 fault
  result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
  return result;
if !IsZero(baseregister<47:40>) then
  level = 0;
  result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
  return result;

// Bottom bound of the Base address is:
//   Log2(8 bytes per entry)+Log2(Num of entries in starting level table)
// Number of entries in starting level table =
//   (Size of Input Address)/(Address per level)^((Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize); // Log2(Num of entries*8)
baseaddress = baseregister<39:baselowerbound>:Zeros(baselowerbound);
ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';
addrselecttop = inputsize - 1;
repeat
  addrselectbottom = (3-level)*stride + grainsize;
  bits(40) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:000');
descaddr.paddress.address = ZeroExtend(baseaddress OR index);
descaddr.paddress.NS = ns_table;
// If there are two stages of translation, then the first stage table walk addresses
// are themselves subject to translation
if secondstage || HasS2Translation() then
descaddr2 = descaddr;
else
    descaddr2 = AArch32.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8);
// Check for a fault on the stage 2 walk
if IsFault(descaddr2) then
    result.addrdesc.fault = descaddr2.fault;
return result;

// Update virtual address for abort functions
descaddr2.vaddress = ZeroExtend(vaddress);

accdesc = CreateAccessDescriptorTTW(acctype, secondstage, s2fs1walk, level);
desc = _Mem[descaddr2, 8, accdesc, iswrite];

if reversedescriptors then desc = BigEndianReverse(desc);
if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
    // Fault (00), Reserved (10), or Block (01) at level 3.
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

// Valid Block, Page, or Table entry
if desc<1:0> == '01' || level == 3 then
    // Block (01) or Page (11)
    blocktranslate = TRUE;
else
    // Table (11)
    if !IsZero(desc<47:40>) then
        result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;

baseaddress = desc<39:grainsize>:Zeros(grainsize);
if !secondstage then
    // Unpack the upper and lower table attributes
    ns_table = ns_table OR desc<63>;
if !secondstage && !hierattrdisabled then
    ap_table<1> = ap_table<1> OR desc<62>; // read-only
    xn_table = xn_table OR desc<60>;
    // pxn_table and ap_table[0] apply only in EL1&0 translation regimes
    if !singlepriv then
        pxn_table = pxn_table OR desc<59>;
        ap_table<0> = ap_table<0> OR desc<61>; // privileged

    level = level + 1;
    addrselecttop = addrselectbottom - 1;
    blocktranslate = FALSE;
until blocktranslate;

// Unpack the descriptor into address and upper and lower block attributes
outputaddress = desc<39:addrselectbottom>:inputaddr<addrselectbottom-1:0>;

// Check the output address is inside the supported range
if !IsZero(desc<47:40>) then
    result.addrdesc.fault = AArch32.AddressSizeFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

// Check the access flag
if desc<10> == '0' then
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

xn = desc<54>;
pxn = desc<53>;
ap = desc<7:6>:'1';
contiguousbit = desc<52>;
\( nG = \text{desc}<11>; \)
\( sh = \text{desc}<9:8>; \)
\( \text{memattr} = \text{desc}<5:2>; \) // AttrIndx and NS bit in stage 1

\( \text{result.domain} = \text{bits}(4) \text{UNKNOWN}; \) // Domains not used
\( \text{result.level} = \text{level}; \)
\( \text{result.blocksize} = 2^\{(3-\text{level})*\text{stride} + \text{grainsize}\}; \)

// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
    \( \text{result.perms.xn} = \text{xn OR \text{xn_table}}; \)
    \( \text{result.perms.ap<2> = ap<2> OR \text{ap_table<1>}}; \) // Force read-only
    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if !singlepriv then
        \( \text{result.perms.ap<1> = ap<1> AND \text{NOT(ap_table<0>)}}; \) // Force privileged only
        \( \text{result.perms.pxn = pxn OR \text{pxn_table}}; \)
        // Pages from Non-secure tables are marked non-global in Secure EL1&0
        if \text{IsSecure}() then
            \( \text{result.nG = nG OR \text{ns_table}}; \)
        else
            \( \text{result.nG = nG}; \)
        endif
    endif
    \( \text{result.GP = desc}<50>; \) // Stage 1 block or pages might be guarded
    \( \text{result.addrdesc.memattrs} = \text{AArch32.S1AttrDecode}(sh, \text{memattr}<2:0>, \text{acctype}); \)
    \( \text{result.addrdesc.paddress.NS} = \text{memattr}<3> \text{OR ns_table}; \)
else
    \( \text{result.perms.ap<2:1> = ap<2:1>}; \)
    \( \text{result.perms.ap<0> = '1'}; \)
    \( \text{result.perms.xn = xn}; \)
    if \text{HaveExtendedExecuteNeverExt}() then \( \text{result.perms.xxxn = desc}<53>; \)
    \( \text{result.perms.pxn = '0'}; \)
    \( \text{result.nG = '0'}; \)
    if s2fs1walk then
        \( \text{result.addrdesc.memattrs} = \text{S2AttrDecode}(sh, \text{memattr}, \text{AccType_TTW}); \)
    else
        \( \text{result.addrdesc.memattrs} = \text{S2AttrDecode}(sh, \text{memattr}, \text{acctype}); \)
        \( \text{result.addrdesc.paddress.NS} = '1'; \)
    endif
    \( \text{result.addrdesc.paddress.address} = \text{ZeroExtend}(\text{outputaddress}); \)
    \( \text{result.addrdesc.fault} = \text{AArch32.NoFault}(); \)
    \( \text{result.contiguous} = \text{contiguousbit} == '1'; \)
    if \text{HaveCommonNotPrivateTransExt}() then \( \text{result.CnP = baseregister<0>}; \)
endif

return result;
// AArch32.TranslationTableWalkSD()
// ---------------------------------------------
// Returns a result of a translation table walk using the Short-descriptor format
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch32.TranslationTableWalkSD(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)
assert ELUsingAArch32(S1TranslationRegime());

// This is only called when address translation is enabled
TLBRecord result;
AddressDescriptor l1descaddr;
AddressDescriptor l2descaddr;
bits(40) outputaddress;

// Variables for Abort functions
ipaddress = bits(40) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;
NS = bit UNKNOWN;

// Default setting of the domain and level.
domain = bits(4) UNKNOWN;
level = 1;

// Determine correct Translation Table Base Register to use.
bvise(64) ttbr;
n = UInt(TTBCR.N);
if n == 0 || IsZero(vaddress<31:(32-n)>) then
  ttbr = TTBR0;
disabled = (TTBCR.PD0 == '1');
else
  ttbr = TTBR1;
disabled = (TTBCR.PD1 == '1');
n = 0; // TTBR1 translation always works like N=0 TTBR0 translation

// Check if Translation table walk disabled for translations with this Base register.
if disabled then
  result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

// Obtain descriptor from initial lookup.
l1descaddr.paddress.address = ZeroExtend(ttbr<31:14-n>:vaddress<31-n:20>:'00');
l1descaddr.paddress.NS = if IsSecure() then '0' else '1';
IRGN = ttbr<0>:ttbr<6>; // TTBR.IRGN
RGN = ttbr<4:3>; // TTBR.RGN
SH = ttbr<1>:ttbr<5>; // TTBR.S:TTBR.NOS
l1descaddr.memattrs = WalkAttrDecode(SH, RGN, IRGN, secondstage);
if !HasS2Translation() then
  // if only 1 stage of translation
  l1descaddr2 = l1descaddr;
else
  l1descaddr2 = AArch32.SecondStageWalk(l1descaddr, vaddress, acctype, iswrite, 4);
  // Check for a fault on the stage 2 walk
  if IsFault(l1descaddr2) then
    result.addrdesc.fault = l1descaddr2.fault;
    return result;

// Update virtual address for abort functions
l1descaddr2.vaddress = ZeroExtend(vaddress);

accdesc = CreateAccessDescriptorTTW(acctype, secondstage, s2fs1walk, level);
l1desc = _Mem[l1descaddr2, 4, accdesc, iswrite];
if SCTLR.EE == '1' then l1desc = BigEndianReverse(l1desc);
// Process descriptor from initial lookup.
case l1desc<1:0> of
  when '00'                                             // Fault, Reserved
    result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, 
iswrite, secondstage, s2fs1walk);
    return result;
  when '01'                                             // Large page or Small page
    domain = l1desc<8:5>;
    level = 2;
    pxn = l1desc<2>;
    NS = l1desc<3>;

    // Obtain descriptor from level 2 lookup.
    l2descaddr.paddress.address = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
    l2descaddr.paddress.NS = if IsSecure() then '0' else '1';
    l2descaddr.memattrs = l1descaddr.memattrs;
    if !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) then
      // if only 1 stage of translation
      l2descaddr2 = l2descaddr;
    else
      l2descaddr2 = AArch32.SecondStageWalk(l2descaddr, vaddress, acctype, iswrite, 4);
      // Check for a fault on the stage 2 walk
      if IsFault(l2descaddr2) then
        result.addrdesc.fault = l2descaddr2.fault;
        return result;
    end;

    // Update virtual address for abort functions
    l2descaddr2.vaddress = ZeroExtend(vaddress);
    accdesc = CreateAccessDescriptorTTW(acctype, secondstage, s2fs1walk, level);
    l2desc = _Mem[l2descaddr2, 4, accdesc, iswrite];
    if SCTLR.EE == '1' then l2desc = BigEndianReverse(l2desc);

    // Process descriptor from level 2 lookup.
    if l2desc<1:0> == '00' then
      result.addrdesc.fault = AArch32.TranslationFault(ipaddress, domain, level, acctype, 
iswrite, secondstage, s2fs1walk);
      return result;
    end;

    nG = l2desc<11>;
    S = l2desc<10>;
    ap = l2desc<9,5:4>;
    if SCTLR.AFE == '1' && l2desc<4> == '0' then
      // Armv8 VMSAv8-32 does not support hardware management of the Access flag.
      result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, 
iswrite, secondstage, s2fs1walk);
      return result;
    end;

    if l2desc<1> == '0' then                           // Large page
      xn = l2desc<15>;
      tex = l2desc<14:12>;
      c = l2desc<3>;
      b = l2desc<2>;
      blocksize = 64;
      outputaddress = ZeroExtend(l2desc<31:16>:vaddress<15:0>);
    else                                               // Small page
      tex = l2desc<8:6>;
      c = l2desc<3>;
      b = l2desc<2>;
      xn = l2desc<0>;
      blocksize = 4;
      outputaddress = ZeroExtend(l2desc<31:12>:vaddress<11:0>);
    end;

  when '1x'                                             // Section or Supersection
    NS = l1desc<19>;

nG = lldesc<17>;  
S = lldesc<16>;  
ap = lldesc<15,11:10>;  
tex = lldesc<14:12>;  
xn = lldesc<4>;  
c = lldesc<3>;  
b = lldesc<2>;  
pxn = lldesc<0>;  
level = 1;

if SCTLR.AFE == '1' && lldesc<10> == '0' then  
  // Armv8 VMSAv8-32 does not support hardware management of the Access flag.  
  result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, 
                                                     iswrite, secondstage, s2fs1walk);  
  return result;

if lldesc<18> == '0' then                          // Section  
  domain = lldesc<8:5>;  
  blocksize = 1024;  
  outputaddress = ZeroExtend(lldesc<31:20>:vaddress<19:0>);  
else                                               // Supersection  
  domain = '0000';  
  blocksize = 16384;  

// Decode the TEX, C, B and S bits to produce the TLBRecord's memory attributes  
if SCTLR.TRE == '0' then  
  if RemapRegsHaveResetValues() then  
    result.addrdesc.memattrs = AArch32.DefaultTEXDecode(tex, c, b, S, acctype);  
  else  
    result.addrdesc.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;  
  else  
    result.addrdesc.memattrs = AArch32.RemappedTEXDecode(tex, c, b, S, acctype);

// Set the rest of the TLBRecord, try to add it to the TLB, and return it.  
result.perms.ap = ap;  
result.perms.xn = xn;  
result.perms.pxn = pxn;  
result.domain = domain;  
result.level = level;  
result.blocksize = blocksize;  
result.addrdesc.paddress.address = ZeroExtend(outputaddress);  
result.addrdesc.paddress.NS = if IsSecure() then NS else '1';  
result.addrdesc.fault = AArch32.NoFault();  
return result;

---

**Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues**

```plaintext
boolean RemapRegsHaveResetValues();
```

---

**Shared Pseudocode Functions**

Page 3086
// BRBEnabled()
// ============
// Returns TRUE if Branch Record Buffer is enabled, FALSE otherwise.

boolean BRBEnabled()
if !HaveBRBExt() then
    return FALSE;
if MDCR_EL3.SBRBE IN {'00', '10'} || (IsSecure() && MDCR_EL3.SBRBE == '01') then
    return TRUE;
if (!EL2Enabled() || HCR_EL2.TGE == '0') && BRBCR_EL1.E0BRE == '1' then
    return TRUE;
if (EL2Enabled() && HCR_EL2.TGE == '1') && BRBCR_EL2.E0HBRE == '1' then
    return TRUE;
if BRBCR_EL1.E1BRE == '1' then
    return TRUE;
if EL2Enabled() && BRBCR_EL2.E2BRE == '1' then
    return TRUE;
return FALSE;

// BRBTimeStamp()
// ==============
// Returns captured timestamp.

TimeStamp BRBTimeStamp()
if HaveEL(EL2) then
    TS_el2 = BRBCR_EL2.TS;
    if !HaveECVExt() && TS_el2 == '10' then
        // Reserved value
        (., TS_el2) = ConstraintUnpredictableBits(Unpredictable_EL2TIMESTAMP);
    case TS_el2 of
        when '00' // Falls out to check BRBCR_EL1.TS
            return TimeStamp.Virtual;
        when '10'
            assert HaveECVExt(); // Otherwise ConstraintUnpredictableBits removes this case
            return TimeStamp.OffsetPhysical;
        when '11'
            return TimeStamp.Physical;
    otherwise
        Unreachable(); // ConstrainUnpredictableBits removes this case
TS_el1 = BRBCR_EL1.TS;
if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
    // Reserved value
    (., TS_el1) = ConstraintUnpredictableBits(Unpredictable_EL1TIMESTAMP);
    case TS_el1 of
        when '01'
            return TimeStamp.Virtual;
        when '10'
            return TimeStamp.OffsetPhysical;
        when '11'
            return TimeStamp.Physical;
    otherwise
        Unreachable(); // ConstrainUnpredictableBits removes this case
Library pseudocode for aarch64/debug/brb/BRB_IALL

```c
// BRB_IALL()
// ==========
// Called to perform invalidation of branch records
BRB_IALL()
    for i = 0 to 63
        Records_SRC[i] = Zeros();
        Records_TGT[i] = Zeros();
        Records_INF[i] = Zeros();
```

Library pseudocode for aarch64/debug/brb/BRB_INJ

```c
// BRB_INJ()
// ==========
// Called to perform manual injection of branch records.
BRB_INJ()
    UpdateBranchRecordBuffer(BRBINFINJ_EL1.CCU, BRBINFINJ_EL1.CC, BRBINFINJ_EL1.LASTFAILED,
                              BRBINFINJ_EL1.T, BRBINFINJ_EL1.TYPE, BRBINFINJ_EL1.EL,
                              BRBINFINJ_EL1.MPREQ, BRBINFINJ_EL1.VALID, BRBSRCINJ_EL1.ADDRESS,
                              BRBTGTINJ_EL1.ADDRESS);
```

Library pseudocode for aarch64/debug/brb/Branch

```c
type BRBSRCType;
type BRBTGTType;
type BRBINFType;
```

Library pseudocode for aarch64/debug/brb/BranchEncCycleCount

```c
// The first return result is '1' if the cycle count is not known, '0' otherwise.
// If the cycle count is known, the second return result is the encoded cycle count since the last branch.
// The format of this field uses a mantissa and exponent to express the cycle count value.
// - bits[7:0] indicate the mantissa M.
// - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
//   cycle_count = (if IsZero(E) then UInt(M) else UInt('1':M:Zeros(UInt(E)-1)))
// A value of all ones in both the mantissa and exponent indicates the cycle count value exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.
(bit, bits(14)) BranchEncCycleCount();
```
Library pseudocode for aarch64/debug/brb/BranchExceptionRecord

// BranchExceptionRecord()
// =======================
// Called to write exception branch record when BRB is active.

BranchExceptionRecord(Exception exception, bits(64) preferred_exception_return, bits(64) target_address, bits(2) target_el)
  if !HaveBRBExt() || !BRBEnabled() then
    return;
  case target_el of
    when EL3 return;
    when EL2 if BRBCR_EL2.EXCEPTION == '0' then return;
    when EL1 if BRBCR_EL1.EXCEPTION == '0' then return;
  boolean sv = BranchRecordAllowed(PSTATE.EL);
  boolean tv = BranchRecordAllowed(target_el);
  if sv || tv then
    bit src_valid = if sv then '1' else '0';
    bit tgt_valid = if tv then '1' else '0';
    UpdateOnException(exception, preferred_exception_return, target_address, target_el, src_valid, tgt_valid);
    return;

Library pseudocode for aarch64/debug/brb/BranchExceptionReturnRecord

// BranchExceptionReturnRecord()
// =============================
// Called to write exception return branch record when BRB is active.

BranchExceptionReturnRecord(bits(64) target_address, bits(2) source_el)
  if !HaveBRBExt() || !BRBEnabled() then
    return;
  case source_el of
    when EL3 return;
    when EL2 if BRBCR_EL2.ERTN == '0' then return;
    when EL1 if BRBCR_EL1.ERTN == '0' then return;
  boolean sv = BranchRecordAllowed(source_el);
  boolean tv = BranchRecordAllowed(PSTATE.EL);
  if sv || tv then
    bit src_valid = if sv then '1' else '0';
    bit tgt_valid = if tv then '1' else '0';
    UpdateOnExceptionReturn(_PC, target_address, PSTATE.EL, src_valid, tgt_valid);
    return;

Library pseudocode for aarch64/debug/brb/BranchMispredict

// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.
boolean BranchMispredict();
Library pseudocode for aarch64/debug/brb/BranchRecord

// BranchRecord()
// ==============
// Called to write non-exception branch record when BRB is active.
BranchRecord(BranchType branch_type, bits(64) target_address)
    if !HaveBRBExt() || !BRBEnabled() then
        return;
    if branch_type IN {BranchType_EXCEPTION, BranchType_ERET, BranchType_DBGEXIT} then
        return;
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(branch_type) then
        UpdateOnNonExceptionBranch(branch_type, _PC, target_address);
        return;

Library pseudocode for aarch64/debug/brb/BranchRecordAllowed

// BranchRecordAllowed()
// ===============
// Returns TRUE if branch recording is allowed, FALSE otherwise.
boolean BranchRecordAllowed(bits(2) el)
    if BRBFRCR_EL1.PAUSED == '1' then
        return FALSE;
    if MDCR_EL3.SBRBE IN {'00', '10'} || (IsSecure() && MDCR_EL3.SBRBE == '01') then
        return FALSE;
    case el of
        when EL3 return FALSE;
        when EL2 return BRBCR_EL2.E2BRE == '1';
        when EL1 return BRBCR_EL1.E1BRE == '1';
        when EL0
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                return BRBCR_EL2.E0HBRE == '1';
            else
                return BRBCR_EL1.E0BRE == '1';

Library pseudocode for aarch64/debug/brb/Contents

array [0..63] of BRBSRCType Records_SRC;
array [0..63] of BRBTGType Records_TGT;
array [0..63] of BRBINFType Records_INF;
Library pseudocode for aarch64/debug/brb/FilterBranchRecord

// FilterBranchRecord()
// ====================
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

boolean FilterBranchRecord(BranchType br)
{
    case br of
        when BranchType_DIRCALL
            return BRBFCR_EL1.DIRCALL != BRBFCR_EL1.EnI;
        when BranchType_INDCALL
            return BRBFCR_EL1.INDCALL != BRBFCR_EL1.EnI;
        when BranchType_RET
            return BRBFCR_EL1.RTN != BRBFCR_EL1.EnI;
        when BranchType_DIR
            return BRBFCR_EL1.DIRECT != BRBFCR_EL1.EnI;
        when BranchType_INDIR
            return BRBFCR_EL1.INDIRECT != BRBFCR_EL1.EnI;
        when BranchType_RESET
            return FALSE;
        when BranchType_UNKNOWN
            return FALSE;
        otherwise
            unreachable();
    return FALSE;
}

Library pseudocode for aarch64/debug/brb/Getter

// Getter functions for branch records
// ===================================
// Functions used by MRS instructions that access branch records

BRBSRCType BRBSRC_EL1[integer n]
{
    assert n IN {0..31};
    return Records_SRC[UInt(BRBFCR_EL1.BANK:n<4:0)];
}

BRBTGTType BRBTGT_EL1[integer n]
{
    assert n IN {0..31};
    return Records_TGT[UInt(BRBFCR_EL1.BANK:n<4:0)];
}

BRBINFType BRBINF_EL1[integer n]
{
    assert n IN {0..31};
    return Records_INF[UInt(BRBFCR_EL1.BANK:n<4:0)];
}

Library pseudocode for aarch64/debug/brb/HaveBRBExt

// HaveBRBExt()
// ============
// Returns TRUE if Branch Record Buffer extension is implemented, FALSE otherwise.

boolean HaveBRBExt()
{
    return boolean IMPLEMENTATION_DEFINED "Has Branch Record Buffer Extension";
}
// UpdateBranchRecordBuffer()  
// ==========================  
// Updates branch record buffer on valid records.

UpdateBranchRecordBuffer(bit ccu, bits(14) cc, bit lastfailed, bit trans, bits(6) branch_type, bits(2) el,  
bit mispredict, bits(2) valid, bits(64) source_address, bits(64) target_address)

// Shift the Branch Records in the buffer
for i = 63 downto 1
    Records_SRC[i] = Records_SRC[i - 1];
    Records_TGT[i] = Records_TGT[i - 1];
    Records_INF[i] = Records_INF[i - 1];

Records_INF[0].CCU        = ccu;
Records_INF[0].CC         = cc;
Records_INF[0].EL         = el;
Records_INF[0].VALID      = valid;
Records_INF[0].T          = trans;
Records_INF[0].LASTFAILED = lastfailed;
Records_INF[0].MPRED      = mispredict;
Records_INF[0].TYPE       = branch_type;

Records_SRC[0] = source_address;
Records_TGT[0] = target_address;
Library pseudocode for aarch64/debug/brb/UpdateOnException

// UpdateOnException()
// =============
// Writes branch record for valid exceptions.

UpdateOnException(Exception exception, bits(64) src_addr, bits(64) tgt_addr,
                    bits(2) target_el, bit src_valid, bit tgt_valid)

    bits(6) branch_type;
    case exception of
        when Exception_Uncategorized branch_type = '100011'; // Trap
        when Exception_WFXTrap branch_type = '100011'; // Trap
        when Exception_CP15RTTrap branch_type = '100011'; // Trap
        when Exception_CP15RRTTrap branch_type = '100011'; // Trap
        when Exception_CP14RTTrap branch_type = '100011'; // Trap
        when Exception_CP14RRTTrap branch_type = '100011'; // Trap
        when Exception_AdvSIMDFPAccessTrap branch_type = '100011'; // Trap
        when Exception_FPIDTrap branch_type = '100011'; // Trap
        when Exception_PACTrap branch_type = '100011'; // Trap
        when Exception_TSTARTAccessTrap branch_type = '100011'; // Trap
        when Exception_CP14RRRTTrap branch_type = '100011'; // Trap
        when Exception_BranchTarget branch_type = '101011'; // Inst Fault
        when Exception_IllegalState branch_type = '100011'; // Trap
        when Exception_SupervisorCall branch_type = '100010'; // Call
        when Exception_HypervisorCall branch_type = '100010'; // Call
        when Exception_MonitorCall branch_type = '100010'; // Call
        when Exception_SystemRegisterTrap branch_type = '100011'; // Trap
        when Exception_SVEAccessTrap branch_type = '100011'; // Trap
        when Exception_FPTrappedException branch_type = '100011'; // Trap
        when Exception_PACFail branch_type = '101100'; // Data Fault
        when Exception_InstructionAbort branch_type = '101011'; // Inst Fault
        when Exception_PCAlignment branch_type = '100101'; // Alignment
        when Exception_DataAbort branch_type = '101100'; // Data Fault
        when Exception_NV2DataAbort branch_type = '101100'; // Data Fault
        when Exception_SPAAlignment branch_type = '101010'; // Alignment
        when Exception_FPTrappedException branch_type = '100011'; // Trap
        when Exception_SError branch_type = '100101'; // System Error
        when Exception_Breakpoint branch_type = '100110'; // Inst debug
        when Exception_SoftwareStep branch_type = '100110'; // Inst debug
        when Exception_Watchpoint branch_type = '100110'; // Data debug
        when Exception_NV2Watchpoint branch_type = '100110'; // Data debug
        when Exception_SoftwareBreakpoint branch_type = '100111'; // Inst debug
        when Exception_IRQ branch_type = '101110'; // IRQ
        when Exception_FIQ branch_type = '101111'; // FIQ
    otherwise
        Unreachable();

    bit ccu;
    bits(14) cc;
    (ccu, cc) = BranchEncCycleCount();
    bit lastfailed = '0';
    bit trans = if HaveTME() && TSTATE.depth > 0 then '1' else '0';
    bits(2) el = target_el;
    bit mispredict = '0';
    bits(64) source_address = if src_valid == '1' then src_addr else Zeros(64);
    bits(64) target_address = if tgt_valid == '1' then tgt_addr else Zeros(64);

    UpdateBranchRecordBuffer(ccu, cc, lastfailed, trans, branch_type, el, mispredict,
                              src_valid: tgt_valid, source_address, target_address);

    return;
Library pseudocode for aarch64/debug/brb/UpdateOnExceptionReturn

// UpdateOnExceptionReturn()
// =========================
// Writes branch record for valid exception returns.

UpdateOnExceptionReturn(bits(64) src_addr, bits(64) tgt_addr, bits(2) target_el,
                       bit src_valid, bit tgt_valid)

  bits(6) branch_type = '000111';
  bit ccu;
  bits(14) cc;
  (ccu, cc) = BranchEncCycleCount();
  bit lastfailed = '0';
  bit trans = if HaveTME() & TSTATE.depth > 0 then '1' else '0';
  bits(2) el = target_el;
  bit mispredict = if BranchMispredict() then '1' else '0';
  bits(64) source_address = if src_valid == '1' then src_addr else
                           Zeros(64);
  bits(64) target_address = if tgt_valid == '1' then tgt_addr else
                           Zeros(64);
  UpdateBranchRecordBuffer(ccu, cc, lastfailed, trans, branch_type, el, mispredict,
                           src_valid:tgt_valid, source_address, target_address);

Library pseudocode for aarch64/debug/brb/UpdateOnNonExceptionBranch

// UpdateOnNonExceptionBranch()
// ============================
// Writes branch record for valid non-exception branches.

UpdateOnNonExceptionBranch(BranchType br_type, bits(64) src_addr, bits(64) tgt_addr)

  bits(6) branch_type;
  case br_type of
    when BranchType_DIR branch_type = '000000';
    when BranchType_INDIR branch_type = '000001';
    when BranchType_DIRCALL branch_type = '000010';
    when BranchType_INDCALL branch_type = '000011';
    when BranchType_RET branch_type = '000101';
    otherwise Unreachable();

  bit ccu;
  bits(14) cc;
  (ccu, cc) = BranchEncCycleCount();
  bit lastfailed = '0';
  bit trans = if HaveTME() & TSTATE.depth > 0 then '1' else '0';
  bits(2) el = PSTATE.EL;
  bit mispredict = if BranchMispredict() then '1' else '0';
  bits(64) source_address = src_addr;
  bits(64) target_address = tgt_addr;
  UpdateBranchRecordBuffer(ccu, cc, lastfailed, trans, branch_type, el, mispredict,
                           '11', source_address, target_address);
Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)
assert !ELUsingAArch32(SJITranslationRegime());
assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

enabled = DBGBCR_EL1[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGBCR_EL1[n].BT == '0x01';
isbreakpnt = TRUE;
linked_to = FALSE;

state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

if HaveAnyAArch32() && size == 4 then                 // Check second halfword
if the breakpoint address and BAS of an Address breakpoint match the address of the
// second halfword of an instruction, but not the address of the first halfword, it is
// CONSTRANGED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
// event.
match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
if !value_match && match_i then
value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
// The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRANGED
// UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
// at the address DBGBVR_EL1[n]+2.
if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

match = value_match && state_match && enabled;
return match;
Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointValueMatch
boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

  // "n" is the identity of the breakpoint unit to match against.
  // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
  // matching breakpoints.
  // "linked_to" is TRUE if this is a call from StateMatch for linking.

  // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
  // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
  if n > UInt(ID_AA64DFR0_EL1.BRPs) then
    (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs), Unpredictable_BPNOTIMPL);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;
  
  // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
  // call from StateMatch for linking).
  if DBGBCR_EL1[n].E == '0' then return FALSE;

  context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));

  // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
  dbgtype = DBGBCR_EL1[n].BT;

  if ((dbgtype IN {'011x','11xx'}) && !HaveVirtHostExt() && !HaveV82Debug()) ||
  dbgtype == '010x' ||
  (dbgtype != '0x0x' && !context_aware) ||
  (dbgtype == '1xxx' && !HaveEL(EL2))) then
    (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;

  // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

  // Determine what to compare against.
  match_addr = (dbgtype == '0x0x');
  match_vmid = (dbgtype == '10xx');
  match_cid = (dbgtype == '001x');
  match_cid1 = (dbgtype IN { '101x', 'x11x'});
  match_cid2 = (dbgtype == '11xx');
  linked = (dbgtype == 'xxx1');

  // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
  // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
  // code can just test for match_addr == TRUE to confirm all these things.
  if linked_to && (!linked || match_addr) then return FALSE;

  // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
  if !linked_to && linked && !match_addr then return FALSE;

  // Do the comparison.
  if match_addr then
    byte = UInt(vaddress<1:0>);
    if HaveAnyAArch32() then
      // T32 instructions can be executed at EL0 in an AArch64 translation regime.
      assert byte IN {0,2}; // "vaddress" is halfword aligned
      byte = (DBGBCR_EL1[n].BAS<byte> == '1');
    else
      assert byte == 0; // "vaddress" is word aligned
      byte = (DBGBCR_EL1[n].BAS<byte> == '1');
  else
    // If 'vaddress' is outside of the current virtual address space, then the access
    // generates a Translation fault.
    integer top = if Have52BitVAExt() then 52 else 48;
    if IsZero(DBGBVREL1[n]<63:top>) && IsZero(DBGBVREL1[n]<63:top>) then
      if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
        top = 63;
BVR_match = (vaddress<top:2> == DBGVR_EL1[n]<top:2>) & byte_select_match;

elsif match_cid then
    if IsInHost() then
        BVR_match = (CONTEXTIDR_EL2<31:0> == DBGVR_EL1[n]<31:0>);
    else
        BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGVR_EL1[n]<31:0>);
    end
elsif match_cid1 then
    BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1<31:0> == DBGVR_EL1[n]<31:0>);
if match_vmid then
    if !Have16bitVmid() || !VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGVR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGVR_EL1[n]<47:32>;
    end
    BXVR_match = (PSTATE.EL IN {EL0, EL1} && !EL2Enabled() && !IsInHost() && vmid == bvr_vmid);
elsif match_cid2 then
    BXVR_match = ((HaveVirtHostExt() || HaveV82Debug()) && EL2Enabled() && DBGVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);
end

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);
match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);
return match;
boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN, boolean isbreakpnt, AccType acctype, boolean ispriv)

// "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
// "linked" is TRUE if this is a linked breakpoint/watchpoint type.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

// If parameters are set to a reserved type, behaves as either disabled or a defined type
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
if c == ConstraintDISABLED then return FALSE;
else if !ispriv && !isbreakpnt then
  priv_match = EL0_match;
else if acctype == AccType_CSR_PRIV && !isbreakpnt then
  assert (PSTATE.EL == EL0) && HaveCSRExt();
  if IsInHost() then
    priv_match = EL2_match;
  else
    priv_match = EL1_match;
  else
    case PSTATE.EL of
      when EL3 priv_match = EL3_match;
      when EL2 priv_match = EL2_match;
      when EL1 priv_match = EL1_match;
      when EL0 priv_match = EL0_match;

    case SSC of
      when '00' security_state_match = TRUE; // Both
      when '01' security_state_match = !IsSecure(); // Non-secure only
      when '10' security_state_match = IsSecure(); // Secure only
      when '11' security_state_match = (HMC == '1' || IsSecure()); // HMC=1 -> Both, 0 -> Secure only
    end
    if linked then
      // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
      // it is CONstrained UNpredictable whether this gives no match, or LBN is mapped to some
      // UNKown breakpoint that is context-aware.
      lbn = UInt(LBN);
      first_ctx_cmp = (UInt(ID AA64DFR0_EL1.BRPs) - UInt(ID AA64DFR0_EL1.CTX_CMPs));
      last_ctx_cmp = UInt(ID AA64DFR0_EL1.BRPs);
      if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
        (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTX);
        assert c IN (ConstraintDISABLED, ConstraintNONE, ConstraintUNKNOWN);
      end
      case c of
        when ConstraintDISABLED return FALSE; // Disabled
        when ConstraintNONE linked = FALSE; // No linking
        // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
      end
    end
    if linked then
      vaddress = bits(64) UNKNOWN;
      linked_to = TRUE;
      linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);
    end
  end
end
return priv_match && security_state_match && (!linked || linked_match);
// AddCallStackRecord()
// ====================
// Generates and then writes a record to Call stack record buffer.

AddCallStackRecord()
  bits(64) ptr;
  bits(6) size;
  bit overflow;

  assert PSTATE.EL != EL3;
  case PSTATE.EL of
    when EL0
      ptr = CSRPTR_EL0.PTR:'000';
      size = CSRCR_EL0.SIZE;
      overflow = CSRCR_EL0.OF;
    when EL1
      ptr = CSRPTR_EL1.PTR:'000';
      size = CSRCR_EL1.SIZE;
      overflow = CSRCR_EL1.OF;
    when EL2
      ptr = CSRPTR_EL2.PTR:'000';
      size = CSRCR_EL2.SIZE;
      overflow = CSRCR_EL2.OF;
    otherwise
      Unreachable();
  endcase;

  integer S;
  if size IN {'001001', '001010', '001011', '001100',
    '001110', '010000', '010101'} then
    S = UInt(size) - 1;
  else
    bits(6) S_unp;
    (-, S_unp) = ConstrainUnpredictableBits(Unpredictable_RESCSRCRSIZE);
    S = UInt(S_unp);
  endcase;

  // Determine if the pointer is saturated
  boolean saturated = IsOnes(ptr<3>);

  // Store the record unless the pointer has Overflown
  if overflow == '0' then
    if PSTATE.EL == EL0 && CallStackRecordingIsPrivileged() then
      acctype = AccType_CSR_PRIV;
    else
      acctype = AccType_CSR_NORMAL;
    endcase;

    // Store the record
    Mem[ptr, 8, acctype] = PC[] + 4;

    // If the pointer was at the Limit, it will now have Overflown
    if saturated then
      overflow = '1';

    // Increment the pointer value
    ptr = ptr + 8;

  case PSTATE.EL of
    when EL0
      CSRPTR_EL0.PTR = ptr<63:3>;
      CSRCR_EL0.OF = overflow;
    when EL1
      CSRPTR_EL1.PTR = ptr<63:3>;
      CSRCR_EL1.OF = overflow;
    when EL2
      CSRPTR_EL2.PTR = ptr<63:3>;
      CSRCR_EL2.OF = overflow;
    otherwise
      Unreachable();
  endcase;
// CSR_PDEC()
// =========
// Decrements Call stack pointer.

CSR_PDEC()
{
    if CallStackRecordingActive()
        then
            RemoveCallStackRecord();

// CallStackRecordingActive()
// ==========================
// Returns TRUE if Call Stack Recording Extension is active.

boolean CallStackRecordingActive()
{
    return CallStackRecordingEnabled() && !CallStackRecordingPaused();

// CallStackRecordingEnabled()
// ===========================
// Returns TRUE if Call Stack Recording Extension is enabled.

boolean CallStackRecordingEnabled()
{
    if UsingAArch32()
        then
            return FALSE;
    case PSTATE.EL of
        when EL0
            then
                EnCSR0 = if IsInHost()
                            then SCTLR_EL2.EnCSR0
                            else SCTLR_EL1.EnCSR0;
                if EnCSR0 == '10'
                    then
                        (., EnCSR0) = ConstrainUnpredictableBits(Unpredictable_RESEnCSR);
        when EL1
            then
                return CSRCR_EL1.EN == '1';
        when EL2
            then
                return CSRCR_EL2.EN == '1';
        when EL3
            then
                return FALSE;

// CallStackRecordingIsPrivileged()
// ================================
// Returns TRUE if Call Stack Recording Extension generates
// privileged memory accesses.

boolean CallStackRecordingIsPrivileged()
{
    EnCSR0 = if PSTATE.EL == EL0
                then
                    EnCSR0 = if IsInHost()
                                then SCTLR_EL2.EnCSR0
                                else SCTLR_EL1.EnCSR0;
                    if EnCSR0 == '10'
                        then
                            (., EnCSR0) = ConstrainUnpredictableBits(Unpredictable_RESEnCSR);
        else
            s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
        else
            s1_enabled = EnCSR0 == '11' && EnCSR0 == '11' && CSRCR_EL0.NP == '0';
    if HasS2Translation()
        then
            s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';

    if s1_enabled
        then
            return TRUE;
    else
        return FALSE;

// CallStackRecordingPaused()
// ==========================
// Returns TRUE if Call Stack Recording Extension is paused.

boolean CallStackRecordingPaused()
{
    return PSTATE.CSR == '0';
}

Library pseudocode for aarch64/debug/csre/RemoveCallStackRecord

// RemoveCallStackRecord()
// =======================
// Decrements Call stack pointer.

RemoveCallStackRecord()
  bits(64) ptr;
  bits(6) size;
  bit underflow;
  bit overflow;

  assert PSTATE.EL != EL3;

case PSTATE.EL of
  when EL0
    ptr = CSRPTR_EL0.PTR:'000';
    size = CSRCR_EL0.SIZE;
    underflow = CSRCR_EL0.UF;
    overflow = CSRCR_EL0.OF;
  when EL1
    ptr = CSRPTR_EL1.PTR:'000';
    size = CSRCR_EL1.SIZE;
    underflow = CSRCR_EL1.UF;
    overflow = CSRCR_EL1.OF;
  when EL2
    ptr = CSRPTR_EL2.PTR:'000';
    size = CSRCR_EL2.SIZE;
    underflow = CSRCR_EL2.UF;
    overflow = CSRCR_EL2.OF;
  otherwise
    Unreachable();

  integer S;
  if size IN {'001001', '001010', '001011', '001100',
              '001110', '010000', '010101'} then
    S = UInt(size) - 1;
  else
    bits(6) S_unp;
    (-,S_unp) = ConstrainUnpredictableBits(Unpredictable_RESCSRCRSIZE);
    S = UInt(S_unp);

  // Determine if the pointer is at the base
  boolean at_base = IsZero(ptr<S:3>);

  // Decrement the pointer value
  if overflow == '0' && !at_base then
    ptr = ptr - 8;
  if at_base then
    underflow = '1';

  case PSTATE.EL of
  when EL0
    CSRPTR_EL0.PTR = ptr<63:3>;
    CSRCR_EL0.UF = underflow;
  when EL1
    CSRPTR_EL1.PTR = ptr<63:3>;
    CSRCR_EL1.UF = underflow;
  when EL2
    CSRPTR_EL2.PTR = ptr<63:3>;
    CSRCR_EL2.UF = underflow;
  otherwise
    Unreachable();

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

// AArch64.GenerateDebugExceptions()
// ----------------------------------------
boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

// AArch64.GenerateDebugExceptionsFrom()
// -----------------------------------------
boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)
    if OSLR_EL1.OSLK == '1' || DoubleLockStatus() || Halted()
        return FALSE;
    route_to_el2 = HaveEL(EL2) && (!secure || IsSecureEL2Enabled()) && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);
    enabled = !HaveEL(EL3) || !secure || MDCR_EL3.SDD == '0';
    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UINT(target) > UINT(from);
    return enabled;

Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

// AArch64.CheckForPMUOverflow()
// ---------------------------------------------
// Signal Performance Monitors overflow IRQ and CTI overflow events
boolean AArch64.CheckForPMUOverflow()
    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1';
    for n = 0 to UINT(PMCR_EL0.N) - 1
        if HaveEL(EL2)
            E = (if n < UINT(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;
    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);
    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)
    return pmuirq;
boolean AArch64.CountEvents(integer n)
assert n == 31 || n < UInt(PMCR_EL0.N);

// Event counting is disabled in Debug state
debug = Halted();

// In Non-secure state, some counters are reserved for EL2
if HaveEL(EL2) then
    resvd_for_el2 = n >= UInt(MDCR_EL2.HPMN) && n != 31;
else
    resvd_for_el2 = FALSE;

// Main enable controls
E = if resvd_for_el2 then MDCR_EL2.HPME else PMCR_EL0.E;
enabled = E == '1' && PMCNTENSET_EL0<n> == '1';

// Event counting is allowed unless it is prohibited by any rule below
prohibited = FALSE;

// Event counting in Secure state is prohibited if all of:
// * EL3 is implemented
// * MDCR_EL3.SPME == 0, and either:
//   - FEAT_PMUv3p7 is not implemented
//   - MDCR_EL3.MPMX == 0
if HaveEL(EL3) && IsSecure() then
    if HavePMUv3p7() then
        prohibited = MDCR_EL3.<SPME,MPMX> == '00';
    else
        prohibited = MDCR_EL3.SPME == '0';

if !prohibited && PSTATE.EL == EL3 && HavePMUv3p7() then
    prohibited = MDCR_EL3.SPME == '1' || !resvd_for_el2;

// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * PMNx is not reserved for EL2
// * MDCR_EL2.HPMD == 1
if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
    prohibited = MDCR_EL2.HPMD == '1';

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// PMCR_EL0.DP disables the cycle counter when event counting is prohibited
if enabled && prohibited && n == 31 then
    enabled = PMCR_EL0.DP == '0';

// If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
// This is not overridden by PMCR_EL0.DP.
if Havev85PMU() && n == 31 then
    if HaveEL(EL3) && IsSecure() && MDCR_EL3.SCCD == '1' then
        prohibited = TRUE;
    if PSTATE.EL == EL2 && MDCR_EL2.HCCD == '1' then
        prohibited = TRUE;

// If FEAT_PMUv3p7 is implemented, cycle counting an be prohibited at EL3.
// This is not overridden by PMCR_EL0.DP.
if HavePMUv3p7() && n == 31 then
    if PSTATE.EL == EL3 && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;
// Event counting might be frozen
frozen = FALSE;

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if HavePMUv3p7() && n != 31 then
  ovflw = PMOVSCLR_EL0<UInt>(PMCR_EL0.N)-1:0>
  if resvd_for_el2 then
    FZ = MDCR_EL2.HPMFZO;
    ovflw<UInt>(MDCR_EL2.HPMN)-1:0> = Zeros();
  else
    FZ = PMCR_EL0.FZO;
    if HaveEL(EL2) then
      ovflw<UInt>(PMCR_EL0.N)-1:UInt>(MDCR_EL2.HPMN)> = Zeros();
    else
      ovflw<UInt>(PMCR_EL0.N)-1:UInt>(MDCR_EL2.HPMN)> = Zeros();
  end
  FZ = MDCR_EL2.HPMFZO;
  ovflw<UInt>(MDCR_EL2.HPMN)-1:0> = Zeros();
else
  ovflw = PMOVSCLR_EL0<UInt>(PMCR_EL0.N)-1:0>
  if resvd_for_el2 then
    FZ = MDCR_EL2.HPMFZO;
    ovflw<UInt>(MDCR_EL2.HPMN)-1:0> = Zeros();
  else
    FZ = PMCR_EL0.FZO;
    if HaveEL(EL2) then
      ovflw<UInt>(PMCR_EL0.N)-1:UInt>(MDCR_EL2.HPMN)> = Zeros();
    else
      ovflw<UInt>(PMCR_EL0.N)-1:UInt>(MDCR_EL2.HPMN)> = Zeros();
  end
  frozen = FZ == '1' && !IsZero(ovflw);

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} bits
filter = if n == 31 then PMCCFILTR_EL0[31:0] else PMEVTYPER_EL0[n]<31:0>;
P = filter<31>;
U = filter<30>;
NSK = if HaveEL(EL3) then filter<29> else '0';
NSU = if HaveEL(EL3) then filter<28> else '0';
NSH = if HaveEL(EL2) then filter<27> else '0';
M = if HaveEL(EL3) then filter<26> else '0';
SH = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';
case PSTATE.EL of
  when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
    when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
    when EL2 filtered = if IsSecure() then NSH == SH else NSH == '0';
    when EL3 filtered = M != P;
return !debug && enabled && !prohibited && !filtered && !frozen;

Library pseudocode for aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess

// CheckProfilingBufferAccess()
// ============================
SysRegAccess CheckProfilingBufferAccess()
if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
  return SysRegAccess_UNDEFINED;
if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.E2PB<0> != '1' then
  return SysRegAccess_TrapToEL2;
if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
  return SysRegAccess_TrapToEL3;
return SysRegAccess_OK;

Library pseudocode for aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess

// CheckStatisticalProfilingAccess()
// ==================================
SysRegAccess CheckStatisticalProfilingAccess()
if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
  return SysRegAccess_UNDEFINED;
if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.TPMS == '1' then
  return SysRegAccess_TrapToEL2;
if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
  return SysRegAccess_TrapToEL3;
return SysRegAccess_OK;
Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

// CollectContextIDR1()
// ---------------------

boolean CollectContextIDR1()
if !StatisticalProfilingEnabled() then return FALSE;
if PSTATE.EL == EL2 then return FALSE;
if EL2Enabled() & HCR_EL2.TGE == '1' then return FALSE;
return PMSR_EL1.CX == '\1';

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

// CollectContextIDR2()
// ---------------------

boolean CollectContextIDR2()
if !StatisticalProfilingEnabled() then return FALSE;
if !EL2Enabled() then return FALSE;
return PMSR_EL2.CX == '\1';

Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

// CollectPhysicalAddress()
// ------------------------

boolean CollectPhysicalAddress()
if !StatisticalProfilingEnabled() then return FALSE;
(secure, el) = ProfilingBufferOwner();
if ((!secure && HaveEL(EL2)) || IsSecureEL2Enabled()) then
  return PMSR_EL2.PA == '\1' && (el == EL2 || PMSR_EL1.PA == '\1');
else
  return PMSR_EL1.PA == '\1';

Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

// CollectTimeStamp()
// ------------------

TimeStamp CollectTimeStamp()
if !StatisticalProfilingEnabled() then return TimeStamp_None;
(secure, el) = ProfilingBufferOwner();
if el == EL2 then
  if PMSR_EL2.TS == '0' then return TimeStamp_None;
else
  if PMSR_EL1.TS == '0' then return TimeStamp_None;
if EL2Enabled() then
  case PMSR_EL2.PCT of
  when '00' return TimeStamp_Virtual;
  when '01' if el == EL2 then return TimeStamp_Physical;
  when '11'
    if (el == EL2 || PMSR_EL1.PCT != '00') && HaveECVExt() then
      return TimeStamp_OffsetPhysical;
    otherwise
      Unreachable();
  case PMSR_EL1.PCT of
  when '00' return TimeStamp_Virtual;
  when '01' return TimeStamp_Physical;
  when '11' if HaveECVExt() then return TimeStamp_OffsetPhysical;
  otherwise
    Unreachable();
Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```c
enumeration OpType {
    OpType_Load,                 // Any memory-read operation other than atomics, compare-and-swap, and swap
    OpType_Store,                // Any memory-write operation, including atomics without return
    OpType_LoadAtomic,           // Atomics with return, compare-and-swap and swap
    OpType_Branch,               // Software write to the PC
    OpType_Other                 // Any other class of operation
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```c
// ProfilingBufferEnabled()
// ================

boolean ProfilingBufferEnabled()
{
    if !HaveStatisticalProfiling() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    non_secure_bit = if secure then '0' else '1';
    return (!ELUsingAArch32(el) && non_secure_bit == SCR_EL3.NS &&
             PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
}
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```c
// ProfilingBufferOwner()
// ================

(boolean, bits(2)) ProfilingBufferOwner()
{
    secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
    el = if !secure && HaveEL(EL2) && MDCR_EL2.E2PB == '00' then EL2 else EL1;
    return (secure, el);
}
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```c
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
```
Library pseudocode for aarch64/debug/statisticalprofiling/SPECollectRecord

boolean SPECollectRecord(bits(64) events, integer total_latency, OpType optype)
assert StatisticalProfilingEnabled();

bits(64) mask = 0xAA<63:0>;                               // Bits [7,5,3,1]
if HaveSVE() then mask<18:17> = Ones();                   // Predicate flags
if HaveTME() then mask<16> = '1';                          // Alignment Flag
if HaveStatisticalProfilingV1p1() then mask<11> = '1';     // Not taken flag
mask<63:48> = bits(16) IMPLEMENTATION_DEFINED;
mask<31:24> = bits(8) IMPLEMENTATION_DEFINED;
mask<15:12> = bits(4) IMPLEMENTATION_DEFINED;

// Check for UNPREDICTABLE case
if (HaveStatisticalProfilingV1p2() && PMSFCR_EL1.<FnE,FE> == '1' &&
    !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
    if ConstrainUnpredictableBool(Unpredictable_BADPMSFCR) then
        return FALSE;
else
    // Filtering by event
    if PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
        e = events AND mask;
        m = PMSEVFR_EL1 AND mask;
        if !IsZero(NOT(e) AND m) then return FALSE;
    // Filtering by inverse event
    if (HaveStatisticalProfilingV1p2() && PMSFCR_EL1.FnE == '1' &&
        !IsZero(PMSEVFR_EL1)) then
        e = events AND mask;
        m = PMSEVFR_EL1 AND mask;
        if !IsZero(e AND m) then return FALSE;
    // Filtering by type
    if PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>) then
        case optype of
            when OpType_Branch
                if PMSFCR_EL1.B == '0' then return FALSE;
            when OpType_Load
                if PMSFCR_EL1.LD == '0' then return FALSE;
            when OpType_Store
                if PMSFCR_EL1.ST == '0' then return FALSE;
            when OpType_LoadAtomic
                if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
            otherwise
                return FALSE;
        // Filtering by latency
        if PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT) then
            if total_latency < UInt(PMSLATFR_EL1.MINLAT) then
                return FALSE;
        // Check for UNPREDICTABLE cases
        if ((PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1 AND mask)) ||
            (PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
            (PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT))) then
            return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        if (HaveStatisticalProfilingV1p2() &&
            ((PMSFCR_EL1.FnE == '1' && !IsZero(PMSEVFR_EL1 AND mask)) ||
            (PMSFCR_EL1.<FnE,FE> == '11' &&
            !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)))) then
            return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
        return TRUE;
Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```java
// StatisticalProfilingEnabled()
// =============================

boolean StatisticalProfilingEnabled()
if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
    return FALSE;

in_host = EL2Enabled() && HCR_EL2.TGE == '1';

secure, el = ProfilingBufferOwner();
if UInt(el) < UInt(PSTATE.EL) || secure != IsSecure() || (in_host && el == EL1) then
    return FALSE;

case PSTATE.EL of
    when EL3 unreachable();
    when EL2 spe_bit = PMSCR_EL2.E2SPE;
    when EL1 spe_bit = PMSCR_EL1.E1SPE;
    when EL0 spe_bit = (if in_host then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);
return spe_bit == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/SysRegAccess

```java
enumeration SysRegAccess {
    SysRegAccess_OK,
    SysRegAccess_UNDEFINED,
    SysRegAccess_TrapToEL1,
    SysRegAccess_TrapToEL2,
    SysRegAccess_TrapToEL3
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```java
enumeration TimeStamp {
    TimeStamp_None,    // No timestamp
    TimeStamp_CoreSight, // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical, // Physical counter value with no offset
    TimeStamp_OffsetPhysical, // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual    // Physical counter value minus CNTVOFF_EL2
};
```
AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
sync_errors = HaveIESB() && SCTLR[target_el].IESB == '1';
if HaveDoubleFaultExt() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && target_el == EL3);
// SCTLR[].IESB might be ignored in Debug state.
if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
if HaveTME() && TSTATE.depth > 0 then
    case exception.exctype of
        when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
        when Exception_Breakpoint cause = TMFailure_DBG;
        when Exception_Watchpoint cause = TMFailure_DBG;
        when Exception_SoftwareStep cause = TMFailure_DBG;
        otherwise cause = TMFailure_ERR;
    FailTransaction(cause, FALSE);
SynchronizeContext();
// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers(target_el);
AArch64.ReportException(exception, target_el);
PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';
SPSR[] = bits(64) UNKNOWN;
ELR[] = bits(64) UNKNOWN;
// PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
PSTATE.IL = '0';
if from_32 then
    // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0';
    // PSTATE.J is RES0
    if (HavePANExt() & (PSTATE.EL == EL1 || (PSTATE.EL == EL2 & ELIsInHost(EL0))) & SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    if HaveMTEExt() then PSTATE.TCO = '1';
    if HaveCSRExt() then PSTATE.CSR = bit UNKNOWN;
DLR_EL0 = bits(64) UNKNOWN;
DSPSR_EL0 = bits(64) UNKNOWN;
EDSCR.ERR = '1';
UpdateEDSCRFields();
// Update EDSCR processor state flags.
if sync_errors then SynchronizeErrors();
EndOfInstruction();
// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n, AccType accType, bits(64) vaddress)

    integer top = if Have52BitVAExt() then 52 else 48;
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;      // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '1111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                     // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASENTINUEOUS);
            bottom = 3;                                        // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED return FALSE;            // Disabled
            when Constraint_NONE mask = 0;                // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

        if mask > bottom then
            // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
            // of DBGVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
            // included in the match.
            if !IsOnes(DBGVR_EL1[n]<63:top>) && !IsZero(DBGVR_EL1[n]<63:top>) then
                if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                    top = 63;
                WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
            // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
            if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
                WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMAKEDBITS);
            else
                WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

        return WVR_match && byte_select_match;
```java
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv, AccType acctype, boolean iswrite)
assert !ELUsingAArch32($1TranslationRegime());
assert n <= UInt(ID_AA64DFR0_EL1.WRP); 

"ispriv" is:
// * FALSE for all loads, stores, and atomic operations executed at EL0.
// * FALSE if the access is unprivileged.
// * TRUE for all other loads, stores, and atomic operations.

enabled = DBGWCR_EL1[n].E == '1';
linked = DBGWCR_EL1[n].WT == '1';
isbreakpnt = FALSE;

state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC, linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
l5_match = FALSE;
if acctype ==AccType_ATOMICRW then
  l5_match = (DBGWCR_EL1[n].LSC != '0');
else
  l5_match = (DBGWCR_EL1[n].LSC<if iswrite then 1 else 0> == '1');

value_match = FALSE;
for byte = 0 to size - 1
  value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);
return value_match && state_match && l5_match && enabled;
```

```java
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)
if IsDebugException(fault) then
  if fault.acctype == AccType_IFETCH then
    if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
      AArch64.VectorCatchException(fault);
    else
      AArch64.BreakpointException(fault);
  else
    AArch64.WatchpointException(vaddress, fault);
  elsif fault.acctype == AccType_IFETCH then
    AArch64.InstructionAbort(vaddress, fault);
  else
    AArch64.DataAbort(vaddress, fault);
```
Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(64) vaddress)
    exception = ExceptionSyndrome(exceptype);
    d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_NV2Watchpoint};
    (exception.syndrome, exception.syndrome2) = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = fault.ipaddress.NS;
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;
    return exception;

Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

// AArch64.CheckPCAlignment()
// ==========================
AArch64.CheckPCAlignment()
    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();

Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

// AArch64.DataAbort()
// ======================
AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR_EL2.TGE == '1' || HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) || HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) || IsSecondStage(fault));
    bits(64) preferred_exception_return = ThisInstrAddr();
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) && IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    else
        exception = AArch64.DataAbort(EL3, exception, preferred_exception_return, vect_offset);
    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCF

// AArch64.EffectiveTCF()
// ======================
// Returns the TCF field applied to tag check faults in the given Exception Level.

bits(2) AArch64.EffectiveTCF(bits(2) el)
  bits(2) tcf;
  if el == EL3 then
    tcf = SCTLR_EL3.TCF;
  elsif el == EL2 then
    tcf = SCTLR_EL2.TCF;
  elsif el == EL1 then
    tcf = SCTLR_EL1.TCF;
  elsif el == EL0 && HCR_EL2.<E2H,TGE> == '11' then
    tcf = SCTLR_EL2.TCF0;
  elsif el == EL0 && HCR_EL2.<E2H,TGE> != '11' then
    tcf = SCTLR_EL1.TCF0;
  return tcf;

Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

// AArch64.InstructionAbort()
// ==========================
AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
// External aborts on instruction fetch must be taken synchronously
if HaveDoubleFaultExt() then assert fault.statuscode != Fault_AsyncExternal;
route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
  (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
   (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault))));

bits(64) preferred_exception_return = ThisInstrAddr();
if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
    IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
  vect_offset = 0x180;
else
  vect_offset = 0x0;

exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
if PSTATE.EL == EL3 || route_to_el3 then
  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
ellif PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
ellif
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.
AArch64.PCAlignmentFault()

  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;

  exception = ExceptionSyndrome(Exception_PCAlignment);
  exception.vaddress = ThisInstrAddr();

  if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
  elsif EL2Enabled() & HCR_EL2.TGE == '1' then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

// AArch64.RaiseTagCheckFault()
// ============================
// Raise a tag check fault exception.
AArch64.RaiseTagCheckFault(bits(64) va, boolean write)

  bits(2) target_el;
  bits(64) preferred_exception_return = ThisInstrAddr();
  integer vect_offset = 0x0;

  if PSTATE.EL == EL0 then
    target_el = if HCR_EL2.TGE == '0' then EL1 else EL2;
  else
    target_el = PSTATE.EL;

  exception = ExceptionSyndrome(Exception_DataAbort);
  exception.syndrome<5:0> = '010001';
  if write then
    exception.syndrome<6> = '1';
    exception.vaddress = bits(4) UNKNOWN : va<59:0>;
  AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

// AArch64.ReportTagCheckFault()
// =============================
// Records a tag check fault exception into the appropriate TCFR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
if el == EL3 then
    assert ttbr == '0';
    TFSR_EL3.TF0 = '1';
elsif el == EL2 then
    if ttbr == '0' then
        TFSR_EL2.TF0 = '1';
    else
        TFSR_EL2.TF1 = '1';
elsif el == EL1 then
    if ttbr == '0' then
        TFSR_EL1.TF0 = '1';
    else
        TFSR_EL1.TF1 = '1';
elsif el == EL0 then
    if ttbr == '0' then
        TFSRE0_EL1.TF0 = '1';
    else
        TFSRE0_EL1.TF1 = '1';

Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAIlignementFault

// AArch64.SPAIignmentFault()
// =========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAIignmentFault()
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;
exception = ExceptionSyndrome(Exception_SPAIignment);
if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif EL2Enabled() & HCR_EL2.TGE == '1' then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/aborts/AArch64.TagCheckFault

// AArch64.TagCheckFault()
// =======================
// Handle a tag check fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccType acctype, boolean iswrite)
  bits(2) tcf = AArch64.EffectiveTCF(PSTATE.EL);
  case tcf of
    when '00'       // Tag Check Faults have no effect on the PE
        return;
    when '01'       // Tag Check Faults cause a synchronous exception
        AArch64.RaiseTagCheckFault(vaddress, iswrite);
    when '10'       // Tag Check Faults are asynchronously accumulated
        AArch64.ReportTagCheckFault(PSTATE.EL, vaddress<55>);
    when '11'       // Tag Check Faults cause a synchronous exception on reads or on
                    // a read-write access, and are asynchronously accumulated on writes
        // Check for access performing both a read and a write.
        readwrite = acctype IN {AccType_ATOMICRW,
                                  AccType_ORDEREDATOMICRW,
                                  AccType_ORDEREDRW};
        if !iswrite || readwrite then
          AArch64.RaiseTagCheckFault(vaddress, iswrite);
        else
          AArch64.ReportTagCheckFault(PSTATE.EL, vaddress<55>);
  end case;

Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

// BranchTargetException()
// =======================
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
  route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_BranchTarget);
  exception.syndrome<1:0> = PSTATE.BTYPE;
  exception.syndrome<24:2> = Zeros();         // RES0
  if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
  elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException

```c
// AArch64.TakePhysicalFIQException()
// ---------------------------------

AArch64.TakePhysicalFIQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
                (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x100;
exception = ExceptionSyndrome(Exception_FIQ);

if route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
odeif PSTATE.EL == EL2 || route_to_el2 then
    assert PSTATE.EL != EL3;
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    assert PSTATE.EL IN {EL0, EL1};
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```c
// AArch64.TakePhysicalIRQException()
// ----------------------------------
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
                (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;
exception = ExceptionSyndrome(Exception_IRQ);

if route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
odeif PSTATE.EL == EL2 || route_to_el2 then
    assert PSTATE.EL != EL3;
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    assert PSTATE.EL IN {EL0, EL1};
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```
Library pseudocode for aarch64/exceptions/asynch/AArch64.TakePhysicalSErrorException

// AArch64.TakePhysicalSErrorException()
// ----------------------------------

AArch64.TakePhysicalSErrorException(boolean impdef_syndrome, bits(24) syndrome)

route_to_el3 = HaveEL3 || SCR_EL3.EA == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x180;

if IsSErrorEdgeTriggered(syndrome) then
    ClearPendingPhysicalSError();

exception = ExceptionSyndrome(Exception_SError);
exception.syndrome<24> = if impdef_syndrome then '1' else '0';
exception.syndrome<23:0> = syndrome;

if PSTATE.EL == EL3 || route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
elsif PSTATE.EL == EL2 || route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

// AArch64.TakeVirtualFIQException()
// ----------------------------------

AArch64.TakeVirtualFIQException()

assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x100;

exception = ExceptionSyndrome(Exception_FIQ);

AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

// AArch64.TakeVirtualIRQException()
// ----------------------------------

AArch64.TakeVirtualIRQException()

assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x80;

exception = ExceptionSyndrome(Exception_IRQ);

AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

```plaintext
// AArch64.TakeVirtualSErrorException()
// -----------------------------------
AArch64.TakeVirtualSErrorException(boolean impdef_syndrome, bits(24) syndrome)
assert PSTATE.EL IN {EL0, EL1} & EL2Enabled();
assert HCR_EL2.TGE == '0' & HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x180;

exception = ExceptionSyndrome(Exception_SError);
if HaveRASExt() then
    exception.syndrome<24> = VSESR_EL2.IDS;
    exception.syndrome<23:0> = VSESR_EL2.ISS;
else
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    if impdef_syndrome then exception.syndrome<23:0> = syndrome;
ClearPendingVirtualSError();
AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

```plaintext
// AArch64.BreakpointException()
// -----------------------------
AArch64.BreakpointException(FaultRecord fault)
assert PSTATE.EL != EL3;

route_to_el2 = (PSTATE.EL IN {EL0, EL1} & EL2Enabled() &
(HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;
vaddress = bits(64) UNKNOWN;
exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);
if PSTATE.EL == EL2 || route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```plaintext
// AArch64.SoftwareBreakpoint()
// -----------------------------
AArch64.SoftwareBreakpoint(bits(16) immediate)
route_to_el2 = (PSTATE.EL IN {EL0, EL1} & EL2Enabled() &
(HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')));

bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
exception.syndrome<15:0> = immediate;
if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```
Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareStepException

// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
assert PSTATE.EL != EL3;

route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x0;

exception = ExceptionSyndrome(Exception_SoftwareStep);
if SoftwareStep_DidNotStep() then
  exception.syndrome<24> = '0';
else
  exception.syndrome<24> = '1';
  exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
 exception.syndrome<5:0> = '100010';                // IFSC = Debug Exception

if PSTATE.EL == EL2 || route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/debug/AArch64.VectorCatchException

// AArch64.VectorCatchException()
// ==============================

// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
assert PSTATE.EL != EL3;
assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

bits(64) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x0;

vaddress = bits(64) UNKNOWN;
exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);
AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
// AArch64.WatchpointException()
// =============================
AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2Watchpoint, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
// AArch64.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer, bit) AArch64.ExceptionClass(Exception exctype, bits(2) target_el)

il_is_valid = TRUE;
from_32 = UsingAArch32();

case exctype of
    when Exception_Uncategorized ec = 0x00; il_is_valid = FALSE;
    when Exception_WFxTrap ec = 0x01;
    when Exception_CP15RTTrap ec = 0x03; assert from_32;
    when Exception_CP15RRTTrap ec = 0x04; assert from_32;
    when Exception_CP14RTTrap ec = 0x05; assert from_32;
    when Exception_CP14DTTrap ec = 0x06; assert from_32;
    when Exception_AdvSIMDFPAccessTrap ec = 0x07;
    when Exception_FPIDTrap ec = 0x08;
    when Exception_PACTrap ec = 0x09;
    when Exception_LDS64BTrap ec = 0x0A;
    when Exception_TSTARTAccessTrap ec = 0x0B;
    when Exception_CP14RRTrap ec = 0x0C; assert from_32;
    when Exception_BranchTarget ec = 0x0D;
    when Exception_IllegalState ec = 0x0E; il_is_valid = FALSE;
    when Exception_SupervisorCall ec = 0x11;
    when Exception_HypervisorCall ec = 0x12;
    when Exception_MonitorCall ec = 0x13;
    when Exception_SystemRegisterTrap ec = 0x18; assert !from_32;
    when Exception_VFEAccessTrap ec = 0x19; assert !from_32;
    when Exception_ERetTrap ec = 0x1A; assert !from_32;
    when Exception_PACFail ec = 0x1C; assert !from_32;
    when Exception_InstructionAbort ec = 0x20; il_is_valid = FALSE;
    when Exception_PCAlignment ec = 0x22; il_is_valid = FALSE;
    when Exception_DataAbort ec = 0x24;
    when Exception_NV2Data Abort ec = 0x25;
    when Exception_SPAignment ec = 0x26; il_is_valid = FALSE; assert !from_32;
    when Exception_FPTrappedException ec = 0x28;
    when Exception_SError ec = 0x2F; il_is_valid = FALSE;
    when Exception_Breakpoint ec = 0x30; il_is_valid = FALSE;
    when Exception_SoftwareStep ec = 0x32; il_is_valid = FALSE;
    when Exception_Watchpoint ec = 0x34; il_is_valid = FALSE;
    when Exception_NV2Watchpoint ec = 0x35; il_is_valid = FALSE;
    when Exception_SoftwareBreakpoint ec = 0x38;
    when Exception_VectorCatch ec = 0x3A; il_is_valid = FALSE; assert from_32;
    otherwise Unreachable();

if ec IN {0x20, 0x24, 0x30, 0x32, 0x34} && target_el == PSTATE.EL then
    ec = ec + 1;

if ec IN {0x11, 0x12, 0x13, 0x28, 0x38} && !from_32 then
    ec = ec + 4;

if il_is_valid then
    il = if ThisInstrLength() == 32 then '1' else '0';
else
    il = '1';
assert from_32 || il == '1'; // AArch64 instructions always 32-bit
return (ec, il);

Shared Pseudocode Functions
Library pseudocode for aarch64/exceptions/exceptions/AArch64.ReportException

// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.
AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;
    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;
    iss2 = exception.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = (Zeros(27) : // <63:37>
                        iss2 :   // <36:32>
                        ec<5:0> : // <31:26>
                        il :     // <25>
                        iss); // <24:0>

    if exceptype IN {Exception/InstructionAbort, Exception(PC/Alignment), Exception/DataAbort,
                     Exception/NV2DataAbort, Exception/NV2Watchpoint,
                     Exception/Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
            if IsSecureEL2Enabled() && IsSecure() then
                HPFAR_EL2.NS = exception.NS;
            else
                HPFAR_EL2.NS = '0';
            HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    return;

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);
// AArch64.TakeReset()
// ================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert !HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;
    fi

    // Reset the system registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';       // Select stack pointer
    PSTATE.<D,A,I,F> = '1111'; // All asynchronous exceptions masked
    PSTATE.SS = '0';       // Clear software step bit
    PSTATE.DIT = '0';      // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IE = '0';       // Clear Illegal Execution state bit
    TSTATE.depth = 0;      // Non-transactional state

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;              // IMPLEMENTATION DEFINED reset vector
    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;
    fi

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:PAMax()>); & IsZero(rv<1:0>);

    BranchTo(rv, BranchType_RESET);
Library pseudocode for aarch64/exceptions/ieeefp/AArch64.FPTrappedException

// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
   exception = ExceptionSyndrome(Exception_FPTrappedException);
   if is_ase then
      if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
         exception.syndrome<23> = '1';                          // TFV
      else
         exception.syndrome<23> = '0';                          // TFV
   else
      exception.syndrome<23> = '1';                              // TFV
   exception.syndrome<10:8> = bits(3) UNKNOWN;                    // VECITR
   if exception.syndrome<23> == '1' then
      exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
   else
      exception.syndrome<7,4:0> = bits(6) UNKNOWN;
   route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';
   bits(64) preferred_exception_return = ThisInstrAddr();
   vect_offset = 0x0;
   if UInt(PSTATE.EL) > UInt(EL1) then
      AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
   elsif route_to_el2 then
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
   else
      AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
   assert HaveEL(EL2);
   if UsingAArch32() then AArch32.ITAdvance();
   SSAdvance();
   bits(64) preferred_exception_return = NextInstrAddr();
   vect_offset = 0x0;
   exception = ExceptionSyndrome(Exception_HypervisorCall);
   exception.syndrome<15:0> = immediate;
   if PSTATE.EL == EL3 then
      AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
   else
      AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

// AArch64.CallSecureMonitor()
// ===========================
AArch64.CallSecureMonitor(bits(16) immediate)
assert HaveEL(EL3) && !ELUsingAArch32(EL3);
if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();
bits(64) preferred_exception_return = NextInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_MonitorCall);
exception.syndrome<15:0> = immediate;
AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor
AArch64.CallSupervisor(bits(16) immediate)
if UsingAArch32() then AArch32.ITAdvance();
SSAdvance();
route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

bits(64) preferred_exception_return = NextInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_SupervisorCall);
exception.syndrome<15:0> = immediate;
if Uint(PSTATE.EL) > Uint(EL1) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
// AArch64.TakeException()
// ==-------------------
// Take an exception to an Exception Level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception, 
bits(64) preferred_exception_return, integer vect_offset)
assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

sync_errors = HaveIESB() && SICTLR[target_el].IESB == '1';
if HaveDoubleFaultExt() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && target_el == EL3);
if sync_errors && InsertIESBBeforeException(target_el) then
    SynchronizeErrors();
    iesb_req = FALSE;
sync_errors = FALSE;
TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

if HaveTME() && TSTATE.depth > 0 then
    case exception.exceptype of
    when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
    when Exception_Breakpoint cause = TMFailure_DBG;
    when Exception_Watchpoint cause = TMFailure_DBG;
    when Exception_SoftwareStep cause = TMFailure_DBG;
    otherwise cause = TMFailure_ERR;
    FailTransaction(cause, FALSE);
    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero from 32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(target_el);
if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    else
        lower_32 = ELUsingAArch32(target_el - 1);
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);
elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;
bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState);
if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
        spsr<3:2> = '10';
    else
        if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
            spsr<3:2> = '10';
    if HaveBTIExt() && !UsingAArch32() then
        // SPSR[].BTYPE is only guaranteed valid for these exception types
        if exception.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
Exception_SoftwareStep, Exception_PCAlignment, Exception_SoftwareAbort, Exception_Breakpoint,
Exception_VectorCatch, Exception_SoftwareBreakpoint, Exception_IllegalState, Exception_BranchTarget} then
            zero_btype = FALSE;
        else
            zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
    if zero_btype then spsr<11:10> = '00';
if HaveNV2Ext() && exception.exceptype == Exception_NV2DataAbort && target_el == EL3 then // external aborts are configured to be taken to EL3 exception.exceptype = Exception_DataAbort;
if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
  AArch64.ReportException(exception, target_el);
BranchExceptionRecord(exception.exceptype, preferred_exception_return, VBAR[target_el]<63:11>:vect_offset<10:0>, target_el);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR[] = spsr;
ELB[] = preferred_exception_return;

PSTATE.SS = '0';
ShouldAdvanceSS = FALSE;
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from_32 then // Coming from AArch32
  PSTATE.IT = '00000000';
PSTATE.T = '0'; // PSTATE.J is RES0
if HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) && 
  SCTLR[].SPAN == '0') then
  PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;
if HaveMTEExt() then PSTATE.TCO = '1';
if HaveCSRExt() then PSTATE.CSR = '0';
BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

if sync_errors then
  SynchronizeErrors();
  iesb_req = TRUE;
  TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
EndOfInstruction();

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

// AArch64.AArch32SystemAccessTrap()
// ==============================================================
// Trapped AARCH32 system register access.
AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
  assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;
  exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
  AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
AArch64.AArch32SystemAccessTrapSyndrome()
// AArch64.AArch32SystemAccessTrapSyndrome()
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
  ExceptionRecord exception;
  case ec of
    when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
    when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
    when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
    when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTrap);
    when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
    when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
    when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
    when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
  otherwise    Unreachable();

  bits(20) iss = Zeros();
  if exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
    // Trapped MRC/MCR, VMRS on FPSID
    if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
      iss<19:17> = instr<7:5>;           // opc2
      iss<16:14> = instr<23:21>;         // opc1
      iss<13:10> = instr<19:16>;         // CRn
      iss<4:1>   = instr<3:0>;           // CRm
    else
      iss<19:17> = '000';
      iss<16:14> = '111';
      iss<13:10> = instr<19:16>;         // reg
      iss<4:1>   = '0000';
    if instr<20> == '1' && instr<15:12> == '1111' then    // MRC, Rt==15
      iss<9:5> = '1111';
    elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
      iss<9:5> = bits(5) UNKNOWN;
    else
      iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    end
  elseif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
    // Trapped MRRC/MCRR, VMRS/VMSR
    if instr<19:16> == '1111' then    // Rt2==15
      iss<14:10> = bits(5) UNKNOWN;
    else
      iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;
    end
    if instr<15:12> == '1111' then    // Rt==15
      iss<9:5> = bits(5) UNKNOWN;
    else
      iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    end
  elseif exception.exceptype == Exception_CP14DTTrap then
    // Trapped LDC/STC
    iss<19:12> = instr<7:0>;         // imm8
    iss<4>   = instr<23>;           // U
    iss<2:1> = instr<24,21>;        // P,W
    if instr<19:16> == '1111' then    // Rn==15, LDC(Literal addressing)/STC
      iss<9:5> = bits(5) UNKNOWN;
      iss<3> = '1';
    else
      exception.exceptype == Exception_Uncategorized then
        // Trapped for unknown reason
        iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
        iss<3> = '0';
      end
      iss<0> = instr<20>;            // Direction
    end
  end
  exception.syndrome<24:20> = ConditionSyndrome();
  exception.syndrome<19:0> = iss;
return exception;

Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

// AArch64.AdvSIMDFPAccessTrap()
// =============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].
AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
    end

return;

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.
boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        major = if nreg == 1 then CRn else CRm;
        if !(IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1' then
            return TRUE;
        end
    end

    // Check for MRC and MCR disabled by HCR_EL2.TIDCP
    if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
        ((CRn == 9 && CRm IN {0,1,2,5,6,7,8}) ||
         (CRn == 10 && CRm IN {0,1,4,8}) ||
         (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15})) then
        return TRUE;
    end

    return FALSE;

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

// AArch64.CheckFPAdvSIMDEnabled()
// ===============================
// Check against CPACR[]
AArch64.CheckFPAdvSIMDEnabled()
if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
    // Check if access disabled in CPACR_EL1
    case CPACR_EL1.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0;
        when '11' disabled = FALSE;
    end
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);
AArch64.CheckFPAdvSIMDTrap(); // Also check against CPTR_EL2 and CPTR_EL3
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()  
  if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
    // Check if access disabled in CPTR_EL2
    if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
      case CPTR_EL2.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
        when '11' disabled = FALSE;
      end if
      if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
    else
      if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
    end if
  if HaveEL(EL3) then
    // Check if access disabled in CPTR_EL3
    if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
  return;

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)  
  route_to_el2 = FALSE;
  // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or HFGITR_EL2.ERET is set,
  // is trapped to EL2
  route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled() &&
    ((HaveNVExt() && HCR_EL2.NV == '1') ||
     (HaveFGTExt() && HCR_EL2.<E2H, TGE> != '11' &&
      (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') && HFGITR_EL2.ERET == '1')));
  if route_to_el2 then
    ExceptionRecord exception;
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = Exception Syndrome(Exception_ERetTrap);
    if eret_with_pac then
      // ERET
      exception.syndrome<1> = '0';
      exception.syndrome<0> = '0';
    else
      exception.syndrome<1> = '1';
      if pac_uses_key_a then
        // ERETAA
        exception.syndrome<0> = '0';
      else
        // ERETAB
        exception.syndrome<0> = '1';
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
if PSTATE.EL == EL0 then UNDEFINED;
if (!PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
    HaveEL(EL3) && SCR_EL3.SMD == '1' then
    UNDEFINED;
route_to_el2 = FALSE;
if !HaveEL(EL3) then
    if PSTATE.EL == EL1 && EL2Enabled() then
        if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
            route_to_el2 = TRUE;
        else
            UNDEFINED;
    else
        UNDEFINED;
else
    route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
if route_to_el2 then
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = imm;
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

// AArch64.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
if HaveFGTExt() then
    route_to_el2 = FALSE;
if PSTATE.EL == EL0 then
    route_to_el2 = (!ELUsingAArch32(EL0) && !ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
    (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));
elsif PSTATE.EL == EL1 then
    route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
    (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));
if route_to_el2 then
    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFxTrap

// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
    if trap then
        AArch64.WFxTrap(target_el, is_wfe);

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_IllegalState);
    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_Uncategorized);
    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

// AArch64.SystemAccessTrap()
// =================================
// Trapped access to AArch64 system register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
assert HaveEL(target_el) & target_el != EL0 & UInt(target_el) >= UInt(PSTATE.EL);
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
ExceptionRecord exception;
case ec of
when 0x0 exception = ExceptionSyndrome(Exception_Uncategorized);
when 0x7 exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
when 0x18 exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
when 0x19 exception = ExceptionSyndrome(Exception_SVEAccessTrap);
otherwise Unreachable();
return exception;

Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

// AArch64.UndefinedFault()
// ========================

AArch64.UndefineedFault()
route_to_el2 = PSTATE.EL == EL0 & EL2Enabled() & HCR_EL2.TGE == '1';
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_Uncategorized);
if UInt(PSTATE.EL) > UInt(EL1) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
elsif route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

// AArch64.WFxTrap()
// ================

AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)
    assert UInt(target_el) > UInt(PSTATE.EL);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<0> = if is_wfe then '1' else '0';
    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();

Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

// CheckLDST64BEnabled()
// =====================
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('10'); // 0x2
    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnALS == '0';
            target_el = if HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnALS == '0';
            target_el = EL2;
        if !trap && ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1) then
            trap = EL2Enabled() && HCRX_EL2.EnALS == '0';
            target_el = EL2;
            if trap then LDST64BTrap(target_el, iss);
// CheckST64BV0Enabled() // ============== // Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
boolean trap = FALSE;
bites(25) iss = ZeroExtend('1'); // 0x1

if PSTATE.EL == EL0 then
  if !IsInHost() then
    trap = SCTLR_EL1.EnAS0 == '0';
    target_el = if HCR_EL2.TGE == '1' then EL2 else EL1;
  else
    trap = SCTLR_EL2.EnAS0 == '0';
    target_el = EL2;
else
  trap = SCTLR_EL2.EnAS0 == '0';
  target_el = EL2;

if !trap && ((PSTATE.EL == EL0 & & !IsInHost()) || PSTATE.EL == EL1) then
  trap = EL2Enabled() & & HCR_EL2.EnAS0 == '0';
  target_el = EL2;

if !trap && ((PSTATE.EL IN {EL0, EL1}) || PSTATE.EL == EL2) then
  trap = HaveEL(EL3) & & SCR_EL3.EnAS0 == '0';
  target_el = EL3;
if trap then LDST64BTrap(target_el, iss);

// CheckST64BVEnabled() // ============== // Checks for trap on ST64BV instruction

CheckST64BVEnabled()
boolean trap = FALSE;
bites(25) iss = Zeros();

if PSTATE.EL == EL0 then
  if !IsInHost() then
    trap = SCTLR_EL1.EnASR == '0';
    target_el = if HCR_EL2.TGE == '1' then EL2 else EL1;
  else
    trap = SCTLR_EL2.EnASR == '0';
    target_el = EL2;
else
  trap = SCTLR_EL2.EnASR == '0';
  target_el = EL2;

if !trap && ((PSTATE.EL == EL0 & & !IsInHost()) || PSTATE.EL == EL1) then
  trap = EL2Enabled() & & HCR_EL2.EnASR == '0';
  target_el = EL2;

if trap then LDST64BTrap(target_el, iss);

// LDST64BTrap() // ============== // Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bites(2) target_el, bites(25) iss)
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = ExceptionSyndrome(Exception_LDST64BTrap);
exception.syndrome = iss;
AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
return;
Library pseudocode for aarch64/exceptions/traps/WFETrapDelay

// WFETrapDelay()
// ==============
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTLR_EL1.TWEDEn == '1';
                delay = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2.TWEDEn == '1';
                delay = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
        when EL2
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay = 1 << (UInt(SRC_EL3.TWEDEL) + 8);
    return (delay_enabled, delay);

Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

// WaitForEventUntilDelay()
// ========================
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay)
    boolean eventarrived = FALSE;
    // set eventarrived to TRUE if WaitForEvent() returns before
    // 'delay' expires when delay_enabled is TRUE.
    return eventarrived;

Library pseudocode for aarch64/functions/aborts/AArch64.CreateFaultRecord

// AArch64.CreateFaultRecord()
// ===========================

FaultRecord AArch64.CreateFaultRecord(Fault statuscode, bits(52) ipaddress, boolean NS,
    integer level, AccType acctype, boolean write, bit extflag,
    bits(2) errortype, boolean secondstage, boolean s2fs1walk)
    FaultRecord fault;
    fault.statuscode = statuscode;
    fault.domain = bits(4) UNKNOWN;         // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN;       // Not used from AArch64
    fault.errortype = errortype;
    fault.ipaddress.NS = if NS then '1' else '0';
    fault.ipaddress.address = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;
    return fault;
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception Level using AArch64.

(bits(25), bits(5)) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;
    bits(25) iss = Zeros();
    bits(5) iss2 = Zeros();

    if (HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 &&
        fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
        iss2 = AArch64.RegisterSpecifier();

    if HaveRASExt() && IsAsyncAbort(fault) then
        if (HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 &&
            fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
            iss<12:11> = '01';  // LST
        else
            iss<12:11> = fault.errortype;  // SET
    if d_side then
        if (IsSecondStage(fault) && !fault.s2fs1walk && (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.acctype == AccType_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk") ) ) then
            iss<24:14> = LSInstructionSyndrome();
        if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
            iss<13> = '1';  // Fault is generated by use of VNCR_EL2
        if fault.acctype IN {AccType_CSR_NORMAL, AccType_CSR_PRIV} then
            assert HaveCSRExt();
            iss<14> = '1';  // Fault is generated by use of Call stack recording
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
        if IsExternalAbort(fault) then iss<9> = fault.extflag;
        iss<7> = if fault.s2fs1walk then '1' else '0';
        iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
        return (iss, iss2);
// AArch64.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusions monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

  // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
  // before or after the check on the local Exclusions monitor. As a result a failure
  // of the local monitor can occur on some implementations even if the memory
  // access would give an memory abort.

  acctype = AccType_ATOMIC;
  iswrite = TRUE;
  aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

  passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
  if !passed then
    return FALSE;

  memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

  if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

  passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
  ClearExclusiveLocal(ProcessorID());

  if passed then
    if memaddrdesc.memattrs.shareable then
      passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

  return passed;

Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);

Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
AArch64.SetExclusiveMonitors() // ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;
    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
        MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
        AArch64.MarkExclusiveVA(address, ProcessorID(), size);

FPRSqrtStepFused() // ==================
// FPRSqrtStepFused(bits(N) op1, bits(N) op2)
bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';                // Use RNE rounding mode
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPFZero(sign);
            else
                result = FPPointF(result_value, fpcr, rounding, fpexc);
            end
        end
    end
    return result;
Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```c
// FPRecipStepFused()
// ===============
bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
assert N IN {16, 32, 64};
bits(N) result;
FPCRType fpcr = FPCR[];
op1 = FPNeg(op1);

boolean altfp = HaveAltFP() && fpcr.AH == '1';
boolean fpexc = !altfp;                         // Generate no floating-point exceptions
if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
   if altfp then fpcr.RMode    = '00';             // Use RNE rounding mode

(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpexc);
FPRounding rounding = FPRoundingMode(fpcr);

if !done then
    inf1  = (type1 == FPType_Infinity);
    inf2  = (type2 == FPType_Infinity);
    zero1 = (type1 == FPType_Zero);
    zero2 = (type2 == FPType_Zero);

    if (inf1 && zero2) || (zero1 && inf2) then
        result = FPTwo('0');
    elsif inf1 || inf2 then
        result = FPInfinity(sign1 EOR sign2);
    else
        // Fully fused multiply-add
        result_value = 2.0 + (value1 * value2);
        if result_value == 0.0 then
            // Sign of exact zero result depends on rounding mode
            sign = if rounding == FPRounding_NEGINF then '1' else '0';
            result = FPZero(sign);
        else
            result = FPRound(result_value, fpcr, rounding, fpexc);
    end
else
    // Sign of exact zero result depends on rounding mode
    result = FPZero(sign);

return result;
```

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

// AArch64.AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;
    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then return FALSE;
    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then return FALSE;
    if !AArch64.AllocationTagAccessIsEnabled(acctype) then return FALSE;
    if acctype IN {AccType_IFETCH, AccType_TTW} then return FALSE;
    if acctype == AccType_NV2REGISTER then return FALSE;
    if acctype IN {AccType_CSR_NORMAL, AccType_CSR_PRIV} then return FALSE;
    if PSTATE.TCO=='1' then return FALSE;
    if !IsTagCheckedInstruction() then return FALSE;
    return TRUE;

Library pseudocode for aarch64/functions/memory/AArch64.AddressWithAllocationTag

// AArch64.AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, AccType acctype, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(acctype) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;

Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

// AArch64.AllocationTagFromAddress()
// ==================================
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;
Library pseudocode for aarch64/functions/memory/AArch64.CheckAlignment

```
// AArch64.CheckAlignment()
// ================

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype, boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW, AccType_ATOMICLS64};
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW, AccType_ATOMICLS64};
    vector  = acctype == AccType_VEC;
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveLSE2Ext() then
        check = (UInt(address<0+:4>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
    else check = atomic || ordered;

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
    return aligned;
```

Library pseudocode for aarch64/functions/memory/AArch64.CheckTag

```
// AArch64.CheckTag()
// ================

// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, bits(4) ptag, boolean write)

    if memaddrdesc.memattrs.tagged then
        return ptag == _MemTag[memaddrdesc];
    else
        return TRUE;
```
library pseudocode for aarch64/functions/memory/AArch64.MemSingle

// AArch64.MemSingle[] - non-assignment (read) form
// ===================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned]
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
bis(size*8) value;
iswrite = FALSE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
if HaveTME() then
    transactional = TSTATE.depth > 0 & & !(acctype IN {AccType_IFETCH, AccType_TTW});
    accdesc = CreateAccessDescriptor(acctype, transactional);
else
    accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);
    value = _Mem[memaddrdesc, size, accdesc, FALSE];
return value;

// AArch64.MemSingle[] - assignment (write) form
// ===================================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) value
assert size IN {1, 2, 4, 8, 16};
assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareable then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
if HaveTME() then
    transactional = TSTATE.depth > 0;
    accdesc = CreateAccessDescriptor(acctype, transactional);
else
    accdesc = CreateAccessDescriptor(acctype);
if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);
    _Mem[memaddrdesc, size, accdesc] = value;
return;
// AArch64.MemTag[] - non-assignment (read) form
// ==================================================
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccType acctype]
assert acctype == AccType_NORMAL;
AddressDescriptor memaddrdesc;
bits(4) value;
iswrite = FALSE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, TRUE, TAG_GRANULE);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Return the granule tag if tagging is enabled...
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
    return _MemTag[memaddrdesc];
else
    // ...otherwise read tag as zero.
    return '0000';

// AArch64.MemTag[] - assignment (write) form
// ==================================================
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccType acctype] = bits(4) value
assert acctype == AccType_NORMAL;
AddressDescriptor memaddrdesc;
iswrite = TRUE;

// Stores of allocation tags must be aligned
if address != Align(address, TAG_GRANULE) then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

wasaligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, TAG_GRANULE);

// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == MemType_Device then
    c = ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE);
    assert c IN {Constraint_NONE, Constraint_FAULT};
    if c == Constraint_FAULT then
        boolean secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
    _MemTag[memaddrdesc] = value;

Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

// AArch64.PhysicalTag()
// =====================
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;
Library pseudocode for aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

// AArch64.TranslateAddressForAtomicAccess()
// ----------------------------------------
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;
    assert size IN {1, 2, 4, 8, 16};
    aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);
    // MMU or MPU lookup
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);
    if HaveMTEExt() && AArch64.AccessIsTagChecked(address, AccType_ATOMICRW) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, AccType_ATOMICRW, iswrite);
    return memaddrdesc;

Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
boolean AddressSupportsLS64(bits(64) address);

Library pseudocode for aarch64/functions/memory/CheckSPAlignment

// CheckSPAlignment()
// ------------------
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    if PSTATE.EL == EL0 then
        stack_align_check = (SCLKR[]).SA0 != '0';
    else
        stack_align_check = (SCLKR[]).SA != '0';
    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();
    return;

Library pseudocode for aarch64/functions/memory/IsBlockDescriptorNTBitValid

// If the implementation supports changing the block size without a break-before-make
// approach, then for implementations that have level 1 or 2 support, the nT bit in
// the block descriptor is valid.
boolean IsBlockDescriptorNTBitValid();

Shared Pseudocode Functions
// Returns True if the current instruction uses tag-checked memory access,
// False otherwise.
boolean IsTagCheckedInstruction();
Library pseudocode for aarch64/functions/memory/Mem
// Mem[] - non-assignment (read) form
// ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
assert size IN {1, 2, 4, 8, 16};
bits(size*8) value;
boolean iswrite = FALSE;

aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
atomic = aligned;
else
  // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
  // 64-bit aligned.
  atomic = address $= Align(address, 8);
if !atomic then
  assert size > 1;
  value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];
  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
    assert c IN {Constraint_FAULT, Constraint_NONE};
    if c == Constraint_NONE then aligned = TRUE;
  for i = 1 to size-1
    value<8*i+7:8*i-1> = AArch64.MemSingle[address+i, 1, acctype, aligned];
  elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
    value<63:0> = AArch64.MemSingle[address, 8, acctype, aligned];
    value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned];
  else
    value = AArch64.MemSingle[address, size, acctype, aligned];
  if BigEndian(acctype) then
    value = BigEndianReverse(value);
return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
boolean iswrite = TRUE;

if BigEndian(acctype) then
value = BigEndianReverse(value);
aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
atomic = aligned;
else
  // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
  // 64-bit aligned.
  atomic = address $= Align(address, 8);
if !atomic then
  assert size > 1;
  AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;
  // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
  // access will generate an Alignment Fault, as to get this far means the first byte did
  // not, so we must be changing to a new translation page.
  if !aligned then
    c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
assert c IN {Constraint_FAULT, Constraint_NONE};
if c == Constraint_NONE then aligned = TRUE;
for i = 1 to size-1
    AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
    AArch64.MemSingle[address, 8, acctype, aligned] = value<63:0>;
    AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>;
else
    AArch64.MemSingle[address, size, acctype, aligned] = value;
return;

Library pseudocode for aarch64/functions/memory/MemAtomic

// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType stacctype)
    bits(size) newvalue;
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);
    // All observers in the shareability domain observe the
    // following load and store atomically.
    oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc, FALSE];
    if BigEndian(ldacctype) then
        oldvalue = BigEndianReverse(oldvalue);
    case op of
        when MemAtomicOp_ADD newvalue = oldvalue + value;
        when MemAtomicOp_BIC newvalue = oldvalue AND NOT(value);
        when MemAtomicOp_EOR newvalue = oldvalue EOR value;
        when MemAtomicOp_ORR newvalue = oldvalue OR value;
        when MemAtomicOp_SMAX newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
        when MemAtomicOp_SMIN newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
        when MemAtomicOp_UMAX newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
        when MemAtomicOp_UMIN newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
        when MemAtomicOp_SWP newvalue = value;
    endcase
    if BigEndian(stacctype) then
        newvalue = BigEndianReverse(newvalue);
    _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;
    // Load operations return the old (pre-operation) value
    return oldvalue;

Library pseudocode for aarch64/functions/memory/MemAtomicCompareAndSwap

```
// MemAtomicCompareAndSwap()
// =========================
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
bits(size) newvalue, AccType ldacctype, AccType stacctype)

    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc, FALSE];
    if BigEndian(ldacctype) then
        oldvalue = BigEndianReverse(oldvalue);
    end if

    if oldvalue == expectedvalue then
        if BigEndian(stacctype) then
            newvalue = BigEndianReverse(newvalue);
        end if
        _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;
    end if

    return oldvalue;
```
Library pseudocode for aarch64/functions/memory/MemLoad64B

```c
// MemLoad64B()
// ============
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccType acctype)
  assert address == Align(address, 64);
  bits(512) data;
  boolean iswrite = FALSE;
  boolean aligned = TRUE;
  if !AddressSupportsLS64(address) then
    c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
    assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};
    if c == Constraint_FAULT then
      // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
      fault = AArch64.ExclusiveFault(acctype, iswrite);
      AArch64.Abort(address, fault);
    else
      // Accesses are not single-copy atomic above the byte level
      for i = 0 to 63
        data<7+8*i : 8*i> = AArch64.MemSingle(address+8*i, 1, acctype, aligned);
      endfor
      return data;
    endif
  endif
  AddressDescriptor memaddrdesc;
  memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, 64);
  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);
  endif
  // Effect on exclusives
  if memaddrdesc.memattrs.shareable then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
  endif
  // Memory array access
  if HaveTME() then
    transactional = TSTATE.depth > 0;
    accdesc = CreateAccessDescriptor(acctype, transactional);
  else
    accdesc = CreateAccessDescriptor(acctype);
  endif
  if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
      bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
      if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
        AArch64.TagCheckFault(address, acctype, iswrite);
      endif
    endif
  endif
  data = Mem[memaddrdesc, 64, accdesc, iswrite];
  return data;
```

Shared Pseudocode Functions
MemStore64B(bits(64) address, bits(512) value, AccType acctype)
    assert address == Align(address, 64);

    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

        if c == Constraint_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            iswrite = TRUE;
            fault = AArch64.ExclusiveFault(acctype, iswrite);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level.
            aligned = TRUE;
            for i = 0 to 63
                AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
        else
            -= MemStore64BWithRet(address, value, acctype);  // Return status is ignored by ST64B
    return;
Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

// MemStore64BWithRet()
// ====================
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccType acctype)
    assert address == Align(address, 64);
    AddressDescriptor memaddrdesc;
    boolean iswrite = TRUE;
    boolean aligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, 64);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
        return ZeroExtend('1');
    
    // Effect on exclusives
    if memaddrdesc.mematts.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);
    
    // Memory array access
    if HaveTME() then
        transactional = TSTATE.depth > 0;
        accdesc = CreateAccessDescriptor(acctype, transactional);
    else
        accdesc = CreateAccessDescriptor(acctype);

    if HaveMTEExt() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
                AArch64.TagCheckFault(address, acctype, iswrite);
                return ZeroExtend('1');
        
        _Mem[memaddrdesc, 64, accdesc] = value;
        status = MemStore64BWithRetStatus();
    return status;

Library pseudocode for aarch64/functions/memory/MemStore64BWithRetStatus

// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.

bits(64) MemStore64BWithRetStatus();
### Library pseudocode for aarch64/functions/memory/NVMem

```
// NVMem[] - non-assignment form
// ================
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
assert offset > 0;
bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
return Mem[address, 8, AccType_NV2REGISTER];
```

```
// NVMem[] - assignment form
// ================
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualisation is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
// * VNCR_EL2.BADDR holds the base address of the memory location, and
// * Offset is the unique offset value defined architecturally for each System register that
// supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
assert offset > 0;
bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
Mem[address, 8, AccType_NV2REGISTER] = value;
return;
```

### Library pseudocode for aarch64/functions/memory/SetTagCheckedInstruction

```
// Flag the current instruction as using/not using memory tag checking.
SetTagCheckedInstruction(boolean checked);
```

### Library pseudocode for aarch64/functions/memory/_MemTag

```
// This _MemTag[] accessor is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an external abort.

bits(4) _MemTag[AddressDescriptor desc, AccessDescriptor accdesc];
```

```
// This _MemTag[] accessor is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The functions address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an external abort.

_MemTag[AddressDescriptor desc, AccessDescriptor accdesc] = bits(4) value;
```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    boolean selbit;
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer top_bit = if tbi then 55 else 63;
    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            
    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);
    
    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);
    
    // Compute the pointer authentication code for a ptr with good extension bits
    if !ischeck then
        ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    
    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);
    
    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if HaveEnhancedPAC() then
            PAC = 0x0000000000000000<63:0>;
        elsif !HaveEnhancedPAC2() then
            PAC<top_bit-1> = NOT(PAC<top_bit-1>);
        
        // preserve the determination between upper and lower address at bit<55> and insert PAC
if !HaveEnhancedPAC2() then
    if tbi then
        result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        if tbi then
            result = ptr<63:56>:selbit:(ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
        else
            result = (ptr<63:56> EOR PAC<63:56>):selbit:(ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
    return result;

Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

// AddPACDA()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) X, bits(64) Y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                    (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
        Enable = SCTLR_EL1.EnDA;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
        Enable = SCTLR_EL2.EnDA;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
        Enable = SCTLR_EL3.EnDA;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;
    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDAKey_EL1, TRUE);
Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

// AddPACDB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APDBKey_EL1;

APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

case PSTATE.EL of
    when EL0
        boolean IsEL1Regime = S1TranslationRegime() == EL1;
        Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
        TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                    (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
        Enable = SCTLR_EL1.EnDB;
        TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
        Enable = SCTLR_EL2.EnDB;
        TrapEL2 = FALSE;
        TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
        Enable = SCTLR_EL3.EnDB;
        TrapEL2 = FALSE;
        TrapEL3 = FALSE;

    if Enable == '0' then return X;
    elsif TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else return AddPAC(X, Y, APDBKey_EL1, TRUE);
// AddPACGA()
// =========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of X, Y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(128) APGAKey_EL1;

APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
case PSTATE.EL of
  when EL0
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0' &&
              (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;
if TrapEL2 then TrapPACUse(EL2);
elsif TrapEL3 then TrapPACUse(EL3);
else return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0><63:32> : Zeros(32);
// AddPACIA()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of X, Y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APIAKey_EL1;

APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = $1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
    TrapEL2 = ($L2Enabled() && HCR_EL2.API == '0' ||
               (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnIA;
    TrapEL2 = $L2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnIA;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnIA;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;
  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return AddPAC(X, Y, APIAKey_EL1, FALSE);
// AddPACIB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication code is
derived using a cryptographic algorithm as a combination of X, Y and the APIBKey_EL1.

bits(64) AddPACIB(bits(64) X, bits(64) Y)
boolean TrapEL2;
boolean TrapEL3;
bits(1) Enable;
bits(128) APIBKey_EL1;

APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = SITranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
               (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnIB;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnIB;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;
if Enable == '0' then return X;
else if TrapEL2 then TrapPACUse(EL2);
else if TrapEL3 then TrapPACUse(EL3);
else return AddPAC(X, Y, APIBKey_EL1, FALSE);

Library pseudocode for aarch64/functions/pac/auth/AArch64.PACFailException

// AArch64.PACFailException()
// ==========================
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
bits(64) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0;

exception = Exception Syndrome(Exception PACFail);
exception.syndrome<1:0> = syndrome;
exception.syndrome<24:2> = Zeros(); // RES0
if UInt(PSTATE.EL) > UInt(EL0) then
  AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
else if route_to_el2 then
  AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
else
  AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number, boolean is_combined);  
bits(64) PAC;  
bits(64) result;  
bits(64) original_ptr;  
bits(2) error_code;  
bits(64) extfield;  

// Reconstruct the extension field used of adding the PAC to the pointer  
boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';  
extfield = Replicate(ptr<55>, 64);  

if tbi then  
  original_ptr = ptr<63:56>:extfield<56-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;  
else  
  original_ptr = extfield<64-bottom_PAC_bit-1:0>:ptr<bottom_PAC_bit-1:0>;  

PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);  
// Check pointer authentication code  
if tbi then  
  if !HaveEnhancedPAC2() then  
    if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then  
      result = original_ptr;  
    else  
      error_code = key_number:NOT(key_number);  
      result = original_ptr<63:55>:error_code:original_ptr<52:0>;  
  else  
    result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;  
    if HaveFPACCombined() || (HaveFPAC() && !is_combined) then  
      if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then  
        error_code = (if data then '1' else '0')::key_number;  
        AArch64.PACFailException(error_code);  
  else  
    if !HaveEnhancedPAC2() then  
      if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then  
        result = original_ptr;  
      else  
        error_code = key_number:NOT(key_number);  
        result = original_ptr<63>:error_code:original_ptr<60:0>;  
    else  
      result = ptr;  
      result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;  
      result<63:56> = result<63:56> EOR PAC<63:56>;  
      if HaveFPACCombined() || (HaveFPAC() && !is_combined) then  
        if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then  
          error_code = (if data then '1' else '0')::key_number;  
          AArch64.PACFailException(error_code);  
    return result;
Library pseudocode for aarch64/functions/pac/authda/AuthDA

// AuthDA()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) X, bits(64) Y, boolean is_combined)
boolean TrapEL2;
boolean TrapEL3;
bites(1) Enable;
bites(128) APDAKey_EL1;

APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
      (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnDA;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnDA;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnDA;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

if Enable == '0' then return X;
elifs TrapEL2 then TrapPACUse(EL2);
elifs TrapEL3 then TrapPACUse(EL3);
else return Auth(X, Y, APDAKey_EL1, TRUE, '0', is_combined);
Library pseudocode for aarch64/functions/pac/authdb/AuthDB

// AuthDB()
// ========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of X, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) X, bits(64) Y, boolean is_combined)
boolean TrapEL2;
boolean TrapEL3;
bits(1)  Enable;
bits(128) APDBKey_EL1;

APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;

case PSTATE.EL of
  when EL0
    boolean IsEL1Regime = S1TranslationRegime() == EL1;
    Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
    TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
      (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL1
    Enable = SCTLR_EL1.EnDB;
    TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL2
    Enable = SCTLR_EL2.EnDB;
    TrapEL2 = FALSE;
    TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
  when EL3
    Enable = SCTLR_EL3.EnDB;
    TrapEL2 = FALSE;
    TrapEL3 = FALSE;

  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else  return Auth(X, Y, APDBKey_EL1, TRUE, '1', is_combined);
// AuthIA()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) X, bits(64) Y, boolean is_combined)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(1) Enable;
  bits(128) APIAKey_EL1;

  APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
  case PSTATE.EL of
    when EL0
      boolean IsEL1Regime = S1TranslationRegime() == EL1;
      Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
      TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL1
      Enable = SCTLR_EL1.EnIA;
      TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL2
      Enable = SCTLR_EL2.EnIA;
      TrapEL2 = FALSE;
      TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
    when EL3
      Enable = SCTLR_EL3.EnIA;
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;
    endcase;

  if Enable == '0' then return X;
  elsif TrapEL2 then TrapPACUse(EL2);
  elsif TrapEL3 then TrapPACUse(EL3);
  else return Auth(X, Y, APIAKey_EL1, FALSE, '0', is_combined);
Library pseudocode for aarch64/functions/pac/authib/AuthIB

// AuthIB()
// =========
// Returns a 64-bit value containing X, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of X, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) X, bits(64) Y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;
        if Enable == '0' then return X;
        elsif TrapEL2 then TrapPACUse(EL2);
        elsif TrapEL3 then TrapPACUse(EL3);
        else return Auth(X, Y, APIBKey_EL1, FALSE, '1', is_combined);
integer CalculateBottomPACBit(bit top_bit)
  integer tsz_field;
  if PtrHasUpperAndLowerAddRanges() then
    assert S1TranslationRegime() IN {EL1, EL2};
    if S1TranslationRegime() == EL1 then
      // EL1 translation regime registers
      tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
      using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
    else
      // EL2 translation regime registers
      assert HaveEL2();
      tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
      using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
  else
    tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
    using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';
  max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if using64k then 47 else 48);
  if tsz_field > max_limit_tsz_field then
    // TCR_ELx.TySZ is out of range
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_NONE};
    if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
      c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
      assert c IN {Constraint_FORCE, Constraint_NONE};
      if c == Constraint_FORCE then tsz_field = tszmin;
  return (64-tsz_field);
Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

array bits(64) RC[0..4];

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
  bits(64) workingval;
  bits(64) runningmod;
  bits(64) roundkey;
  bits(64) modk0;
  constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;
  RC[0] = 0x0000000000000000<63:0>;
  RC[1] = 0x13198A2E03797344<63:0>;
  RC[2] = 0xA4093822299F31D0<63:0>;
  RC[3] = 0x082EFA98EC4E6C89<63:0>;
  RC[4] = 0x452821E63801377<63:0>;

  modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
  runningmod = modifier;
  workingval = data EOR key0;
  for i = 0 to 4
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = workingval EOR RC[i];
    if i > 0 then
      workingval = PACCellShuffle(workingval);
      workingval = PACMult(workingval);
      runningmod = PACSub(runningmod);
      roundkey = modk0 EOR runningmod;
    end if
  end for
  return workingval;

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

// PACCellInvShuffle()
// =============

bits(64) PACCellInvShuffle(bits(64) indata)
  bits(64) outdata;
  outdata<3:0> = indata<15:12>;
  outdata<7:4> = indata<27:24>;
  outdata<11:8> = indata<51:48>;
  outdata<15:12> = indata<39:36>;
  outdata<19:16> = indata<59:56>;
  outdata<23:20> = indata<47:44>;
  outdata<27:24> = indata<7:4>;
  outdata<31:28> = indata<19:16>;
  outdata<35:32> = indata<35:32>;
  outdata<39:36> = indata<55:52>;
  outdata<43:40> = indata<31:28>;
  outdata<47:44> = indata<11:8>;
  outdata<51:48> = indata<23:20>;
  outdata<55:52> = indata<3:0>;
  outdata<59:56> = indata<43:40>;
  outdata<63:60> = indata<63:60>;
  return outdata;

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

// PACCellShuffle()
// ===============

bits(64) PACCellShuffle(bits(64) indata)
  bits(64) outdata;
  outdata<3:0> = indata<55:52>;
  outdata<7:4> = indata<27:24>;
  outdata<11:8> = indata<47:44>;
  outdata<15:12> = indata<3:0>;
  outdata<19:16> = indata<31:28>;
  outdata<23:20> = indata<51:48>;
  outdata<27:24> = indata<7:4>;
  outdata<31:28> = indata<43:40>;
  outdata<35:32> = indata<35:32>;
  outdata<39:36> = indata<55:52>;
  outdata<43:40> = indata<31:28>;
  outdata<47:44> = indata<11:8>;
  outdata<51:48> = indata<23:20>;
  outdata<55:52> = indata<3:0>;
  outdata<59:56> = indata<43:40>;
  outdata<63:60> = indata<63:60>;
  return outdata;
Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
  // This is a 4-bit substitution from the PRINCE-family cipher

  bits(64) Toutput;
  for i = 0 to 15
    case Tinput<4*i+3:4*i> of
      when '0000'  Toutput<4*i+3:4*i> = '0101';
      when '0001'  Toutput<4*i+3:4*i> = '1110';
      when '0010'  Toutput<4*i+3:4*i> = '1101';
      when '0011'  Toutput<4*i+3:4*i> = '1000';
      when '0100'  Toutput<4*i+3:4*i> = '1010';
      when '0101'  Toutput<4*i+3:4*i> = '1011';
      when '0110'  Toutput<4*i+3:4*i> = '0001';
      when '0111'  Toutput<4*i+3:4*i> = '1001';
      when '1000'  Toutput<4*i+3:4*i> = '0010';
      when '1001'  Toutput<4*i+3:4*i> = '0110';
      when '1010'  Toutput<4*i+3:4*i> = '1111';
      when '1011'  Toutput<4*i+3:4*i> = '0000';
      when '1100'  Toutput<4*i+3:4*i> = '0100';
      when '1101'  Toutput<4*i+3:4*i> = '1100';
      when '1110'  Toutput<4*i+3:4*i> = '0111';
      when '1111'  Toutput<4*i+3:4*i> = '0011';
    endcase
  return Toutput;

Library pseudocode for aarch64/functions/pac/computepac/PACMult

// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)

  bits(4)  t0;
  bits(4)  t1;
  bits(4)  t2;
  bits(4)  t3;
  bits(64) Soutput;

  for i = 0 to 3
    t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
    t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
    t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
    t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
    t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    Soutput<4*i+3:4*i> = t3<3:0>;
    Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
    Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
    Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
  return Soutput;
Library pseudocode for aarch64/functions/pac/computepac/PACSub

// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Tinput<4*i+3:4*i> of
        when '0000'  Toutput<4*i+3:4*i> = '1011';
        when '0001'  Toutput<4*i+3:4*i> = '0110';
        when '0010'  Toutput<4*i+3:4*i> = '1000';
        when '0011'  Toutput<4*i+3:4*i> = '1111';
        when '0100'  Toutput<4*i+3:4*i> = '1100';
        when '0101'  Toutput<4*i+3:4*i> = '0000';
        when '0110'  Toutput<4*i+3:4*i> = '1001';
        when '0111'  Toutput<4*i+3:4*i> = '1110';
        when '1000'  Toutput<4*i+3:4*i> = '0011';
        when '1001'  Toutput<4*i+3:4*i> = '0111';
        when '1010'  Toutput<4*i+3:4*i> = '0100';
        when '1011'  Toutput<4*i+3:4*i> = '0101';
        when '1100'  Toutput<4*i+3:4*i> = '1101';
        when '1101'  Toutput<4*i+3:4*i> = '0010';
        when '1110'  Toutput<4*i+3:4*i> = '0001';
        when '1111'  Toutput<4*i+3:4*i> = '1010';
    endcase;
return Toutput;

Library pseudocode for aarch64/functions/pac/computepac/RotCell

// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
bits(8) tmp;
bits(4) outcell;

// assert amount>3 || amount<1;
tmp<7:0> = incell<3:0>:incell<3:0>;
outcell = tmp<7-amount:4-amount>;
return outcell;

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

// TweakCellInvRot()
// ===============

bits(4) TweakCellInvRot(bits(4) incell)
bits(4) outcell;
outcell<3> = incell<2>;
outcell<2> = incell<1>;
outcell<1> = incell<0>;
outcell<0> = incell<0> EOR incell<3>;
return outcell;

Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
bits(4) outcell;
outcell<3> = incell<0> EOR incell<1>;
outcell<2> = incell<3>;
outcell<1> = incell<2>;
outcell<0> = incell<1>;
return outcell;
Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

// TweakInvShuffle()
// ================

bits(64) TweakInvShuffle(bits(64) indata)
   bits(64) outdata;
   outdata<3:0> = TweakCellInvRot(indata<51:48>);
   outdata<7:4> = indata<55:52>;
   outdata<11:8> = indata<23:20>;
   outdata<15:12> = TweakCellInvRot(indata<27:24>);
   outdata<19:16> = indata<3:0>;
   outdata<23:20> = indata<7:4>;
   outdata<27:24> = TweakCellInvRot(indata<11:8>);
   outdata<31:28> = indata<15:12>;
   outdata<35:32> = TweakCellInvRot(indata<31:28>);
   outdata<39:36> = TweakCellInvRot(indata<63:60>);
   outdata<43:40> = TweakCellInvRot(indata<59:56>);
   outdata<47:44> = TweakCellInvRot(indata<19:16>);
   outdata<51:48> = indata<35:32>;
   outdata<55:52> = indata<39:36>;
   outdata<59:56> = indata<43:40>;
   outdata<63:60> = TweakCellInvRot(indata<47:44>);
   return outdata;

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
   bits(64) outdata;
   outdata<3:0> = indata<19:16>;
   outdata<7:4> = indata<23:20>;
   outdata<11:8> = TweakCellRot(indata<27:24>);
   outdata<15:12> = indata<31:28>;
   outdata<19:16> = TweakCellRot(indata<47:44>);
   outdata<23:20> = indata<11:8>;
   outdata<27:24> = TweakCellRot(indata<15:12>);
   outdata<31:28> = TweakCellRot(indata<35:32>);
   outdata<35:32> = TweakCellRot(indata<51:48>);
   outdata<39:36> = indata<55:52>;
   outdata<43:40> = indata<59:56>;
   outdata<47:44> = TweakCellRot(indata<63:60>);
   outdata<51:48> = TweakCellRot(indata<3:0>);
   outdata<55:52> = TweakCellRot(indata<7:4>);
   outdata<59:56> = TweakCellRot(indata<43:40>);
   outdata<63:60> = TweakCellRot(indata<39:36>);
   return outdata;

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC

/// HaveEnhancedPAC()
/// ================
/// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
   return ( HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality" );

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC2

/// HaveEnhancedPAC2()
/// ===============
/// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
   return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC 2 functionality");
Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

```plaintext
// HaveFPAC()
// ==========
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return HaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPACCombined

```plaintext
// HaveFPACCombined()
// ================
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return HaveFPAC() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```plaintext
// HavePACExt()
// ============
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```plaintext
// PtrHasUpperAndLowerAddRanges()
// =============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    return PSTATE.EL == EL1 || PSTATE.EL == EL0 || (PSTATE.EL == EL2 && HCR_EL2.E2H == '1');
```

Library pseudocode for aarch64/functions/pac/strip/Strip

```plaintext
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
code field bits with the extension of the address bits. This can apply to either
instructions or data, where, as the use of tagged pointers is distinct, it might be
handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);
    if tbi then
        original_ptr = A<63:56>:extfield< 56-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield< 64-bottom_PAC_bit-1:0>:A<bottom_PAC_bit-1:0>;
    return original_ptr;
```
Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception levels.
TrapPACUse(bits(2) target_el) {
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);
    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
}

Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

// AArch64.ESBOperation()
// =============
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for ESB in AArch32 state when SError interrupts are routed to an Exception level using AArch64
AArch64.ESBOperation()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;
    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;
    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
                                    PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSErrorSyndrome();
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
            ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0
        return;
    return;

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalLErrorSyndrome

// Return the SError syndrome
bits(25) AArch64.PhysicalLErrorSyndrome(boolean implicit_esb);
Library pseudocode for aarch64/functions/ras/AArch64.ReportDeferredSError

// AArch64.ReportDeferredSError()
// ==============================
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
bits(64) target;
target<31> = '1'; // A
target<24> = syndrome<24>; // IDS
target<23:0> = syndrome<23:0>; // ISS
return target;

Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
// If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
// SError interrupt might be pending
vSEI enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
vSEI pending = vSEI enabled && HCR_EL2.VSE == '1';
vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
vmasked = dintdis || PSTATE.A == '1';
// Check for a masked virtual SError pending
if vSEI pending & vmasked then
    // This function might be called for the interworking case, and INTdis is masking
    // the virtual SError interrupt.
    if ELUsingAArch32(EL1) then
        VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
    else
        VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
    HCR_EL2.VSE = '0';                       // Clear pending virtual SError
return;

Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
assert UsingAArch32();                     // Always called from AArch32 state before entering AArch64 state
if PSTATE.EL == EL0 && ELUsingAArch32(EL1) then
    first = 0; last = 14; include_R15 = FALSE;
else if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && ELUsingAArch32(EL2) then
    first = 0; last = 30; include_R15 = FALSE;
else
    first = 0; last = 30; include_R15 = TRUE;
for n = first to last
    if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
        _R[n]<63:32> = Zeros();
return;
Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

// AArch64.ResetGeneralRegisters()
// ===============================
AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;

Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

// AArch64.ResetSIMDFPRegisters()
// ==============================
AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;

Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

// AArch64.ResetSpecialRegisters()
// ===============================
AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;

Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

AArch64.ResetSystemRegisters(boolean cold_reset);
Library pseudocode for aarch64/functions/registers/PC

// PC - non-assignment form
// --------------------------
// Read program counter.

bits(64) PC[]
    return _PC;

Library pseudocode for aarch64/functions/registers/SP

// SP[] - assignment form
// ------------------------
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
    assert width IN {32,64};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    else
        case PSTATE.EL of
            when EL0 SP_EL0 = ZeroExtend(value);
            when EL1 SP_EL1 = ZeroExtend(value);
            when EL2 SP_EL2 = ZeroExtend(value);
            when EL3 SP_EL3 = ZeroExtend(value);
        return;

Library pseudocode for aarch64/functions/registers/V

// V[] - assignment form
// ---------------------
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    integer vlen = if IsSVEEnabled(PSTATE.EL) then VL else 128;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value);

Library pseudocode for aarch64/functions/registers/SP

// SP[] - non-assignment form
// --------------------------
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0 return SP_EL0<width-1:0>;
            when EL1 return SP_EL1<width-1:0>;
            when EL2 return SP_EL2<width-1:0>;
            when EL3 return SP_EL3<width-1:0>;
    return;

Library pseudocode for aarch64/functions/registers/V

// V[] - non-assignment form
// --------------------------
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _Z[n]<width-1:0>;
Library pseudocode for aarch64/functions/registers/Vpart

// Vpart[] - non-assignment form
// =============================
// Reads a 128-bit SIMD&FP register in up to two parts:
// part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
// part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
// value held in the register.

bits(width) Vpart[integer n, integer part]
  assert n >= 0 && n <= 31;
  assert part IN {0, 1};
  if part == 0 then
    assert width < 128;
    return V[n];
  else
    assert width IN {32,64};
    bits(128) vreg = V[n];
    return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =========================
// Writes a 128-bit SIMD&FP register in up to two parts:
// part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
// part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part] = bits(width) value
  assert n >= 0 && n <= 31;
  assert part IN {0, 1};
  if part == 0 then
    assert width < 128;
    V[n] = value;
  else
    assert width == 64;
    bits(64) vreg = V[n];
    V[n] = value<63:0> : vreg;

Library pseudocode for aarch64/functions/registers/X

// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X[integer n] = bits(width) value
  assert n >= 0 && n <= 31;
  assert width IN {32,64};
  if n != 31 then
    _R[n] = ZeroExtend(value);
    return;

// X[] - non-assignment form
// =========================
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
  assert n >= 0 && n <= 31;
  assert width IN {8,16,32,64};
  if n != 31 then
    return _R[n]<width-1:0>;
  else
    return Zeros(width);
// AArch32.IsFPEnabled()
// --------------------
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
if el == EL0 && !ELUsingAArch32(EL1) then
    return AArch64.IsFPEnabled(el);
if HaveEL(EL3) && ELUsingAArch32(EL3) && 'IsSecure() then
    // Check if access disabled in NSACR
    if NSACR.cp10 == '0' then return FALSE;
if el IN {EL0, EL1} then
    // Check if access disabled in CPACR
    case CPACR.cp10 of
        when '00' disabled = TRUE;
        when '01' disabled = el == EL0;
        when '10' disabled = ConstranUnpredictableBool(Unpredictable_RESCPACR);
        when '11' disabled = FALSE;
    if disabled then return FALSE;
if el IN {EL0, EL1, EL2} && EL2Enabled() then
    if !ELUsingAArch32(EL2) then
        return AArch64.IsFPEnabled(EL2);
    if HCPTR.TCP10 == '1' then return FALSE;
if HaveEL(EL3) && 'ELUsingAArch32(EL3) then
    // Check if access disabled in CPTR_EL3
    if CPTR_EL3.TFP == '1' then return FALSE;
return TRUE;

// AArch64.IsFPEnabled()
// --------------------
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
// Check if access disabled in CPACR_EL1
if el IN {EL0, EL1} && !IsInHost() then
    // Check FP&SIMD at EL0/EL1
    case CPACR_EL1.FPEN of
        when 'x0' disabled = TRUE;
        when '01' disabled = el == EL0;
        when '11' disabled = FALSE;
    if disabled then return FALSE;
// Check if access disabled in CPTR_EL2
if el IN {EL0, EL1, EL2} && EL2Enabled() then
    if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
        case CPTR_EL2.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
            when '11' disabled = FALSE;
        if disabled then return FALSE;
    else
        if CPTR_EL2.TFP == '1' then return FALSE;
// Check if access disabled in CPTR_EL3
if HaveEL(EL3) then
    if CPTR_EL3.TFP == '1' then return FALSE;
return TRUE;
Library pseudocode for aarch64/functions/sve/BitDeposit

// BitDeposit()
// ============
// Deposit the least significant bits from DATA into result positions
// selected by non-zero bits in MASK, setting other result bits to zero.

bits(N) BitDeposit (bits(N) data, bits(N) mask)
    bits(N) res = Zeros();
    integer db = 0;
    for rb = 0 to N-1
        if mask<rb> == '1' then
            res<rb> = data<db>;
            db = db + 1;
    return res;

Library pseudocode for aarch64/functions/sve/BitExtract

// BitExtract()
// ============
// Extract and pack DATA bits selected by the non-zero bits in MASK into
// the least significant result bits, setting other result bits to zero.

bits(N) BitExtract (bits(N) data, bits(N) mask)
    bits(N) res = Zeros();
    integer rb = 0;
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;

Library pseudocode for aarch64/functions/sve/BitGroup

// BitGroup()
// ============
// Extract and pack DATA bits selected by the non-zero bits in MASK into
// the least significant result bits, and pack unselected bits into the
// most significant result bits.

bits(N) BitGroup (bits(N) data, bits(N) mask)
    bits(N) res;
    integer rb = 0;

    // compress masked bits to right
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    // compress unmasked bits to left
    for db = 0 to N-1
        if mask<db> == '0' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;

Library pseudocode for aarch64/functions/sve/CeilPow2

// CeilPow2()
// ===========
// For a positive integer \( X \), return the smallest power of 2 \( \geq X \)

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
Library pseudocode for aarch64/functions/sve/CheckSVEEnabled

// CheckSVEEnabled()
// ================
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
// Check if access disabled in CPACR_EL1
if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
  // Check SVE at EL0/EL1
  case CPACR_EL1.ZEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = PSTATE.EL == EL0;
    when '11' disabled = FALSE;
  if disabled then SVEAccessTrap(EL1);

  // Check SIMD&FP at EL0/EL1
  case CPACR_EL1.FPEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = PSTATE.EL == EL0;
    when '11' disabled = FALSE;
  if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

// Check if access disabled in CPTR_EL2
if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
  if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
    // Check SVE at EL2
    case CPTR_EL2.ZEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
      when '11' disabled = FALSE;
    if disabled then SVEAccessTrap(EL2);

    // Check SIMD&FP at EL2
    case CPTR_EL2.FPEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
      when '11' disabled = FALSE;
    if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
  else
    if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
    if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
  endif

// Check if access disabled in CPTR_EL3
if HaveEL(EL3) then
  if CPTR_EL3.EZ == '0' then SVEAccessTrap(EL3);
  if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
library pseudocode for aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// ================

integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = VL DIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise numElem = 0;
    return numElem;
```

library pseudocode for aarch64/functions/sve/ElemFFR

```
// ElemFFR[] - non-assignment form
// ===============================

bit ElemFFR[integer e, integer esize]
    return ElemP[FFR, e, esize];

// ElemFFR[] - assignment form
// ===========================

ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= PL;
    _FFR<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```

library pseudocode for aarch64/functions/sve/ElemP

```
// ElemP[] - non-assignment form
// =============================

bit ElemP[bits(N) pred, integer e, integer esize]
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;

// ElemP[] - assignment form
// =========================

ElemP[bits(N) &pred, integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= N;
    pred<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```
Library pseudocode for aarch64/functions/sve/FFR

// FFR[] - non-assignment form
// ============================
bits(width) FFR[]
    assert width == PL;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =======================
FFR[] = bits(width) value
    assert width == PL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _FFR = ZeroExtend(value);
    else
        _FFR<width-1:0> = value;

Library pseudocode for aarch64/functions/sve/FPCompareNE

// FPCompareNE()
// =============
boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    if op1_nan || op2_nan then
        result = TRUE;
    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);
    return result;

Library pseudocode for aarch64/functions/sve/FPCompareUN

// FPCompareUN()
// =============
boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
    result = type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN};
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);
    return result;
Library pseudocode for aarch64/functions/sve/FPConvertSVE

```c
// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding);

// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr)
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for aarch64/functions/sve/FPExpA

```c
// FPExpA()
// ========

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;
    return result;
```
// FPExpCoefficient()
// ===============

bits(N) FPExpCoefficient[integer index]
assert N IN \{16,32,64\};
integer result;

if N == 16 then
  case index of
    when 0 result = 0x0000;
    when 1 result = 0x0016;
    when 2 result = 0x002d;
    when 3 result = 0x0045;
    when 4 result = 0x005d;
    when 5 result = 0x0075;
    when 6 result = 0x008e;
    when 7 result = 0x00a8;
    when 8 result = 0x00c2;
    when 9 result = 0x00dc;
    when 10 result = 0x00f8;
    when 11 result = 0x0114;
    when 12 result = 0x0130;
    when 13 result = 0x014d;
    when 14 result = 0x016b;
    when 15 result = 0x0189;
    when 16 result = 0x01a8;
    when 17 result = 0x01c8;
    when 18 result = 0x01e8;
    when 19 result = 0x0209;
    when 20 result = 0x022b;
    when 21 result = 0x024e;
    when 22 result = 0x0271;
    when 23 result = 0x0295;
    when 24 result = 0x02ba;
    when 25 result = 0x02e0;
    when 26 result = 0x0306;
    when 27 result = 0x032e;
    when 28 result = 0x0356;
    when 29 result = 0x037f;
    when 30 result = 0x03a9;
    when 31 result = 0x03d4;
  elsif N == 32 then
    case index of
      when 0 result = 0x000000;
      when 1 result = 0x0164d2;
      when 2 result = 0x02cd87;
      when 3 result = 0x043a29;
      when 4 result = 0x05aac3;
      when 5 result = 0x071f62;
      when 6 result = 0x08980f;
      when 7 result = 0x0a14d5;
      when 8 result = 0x0b95c2;
      when 9 result = 0x0d1adf;
      when 10 result = 0xeae43a;
      when 11 result = 0x1031dc;
      when 12 result = 0x11c3d3;
      when 13 result = 0x135a2b;
      when 14 result = 0x14f4f0;
      when 15 result = 0x16942d;
      when 16 result = 0x1837f0;
      when 17 result = 0x19e046;
      when 18 result = 0x1be8d3a;
      when 19 result = 0x1d3eda;
      when 20 result = 0x1ef532;
      when 21 result = 0x20b051;
      when 22 result = 0x227043;
      when 23 result = 0x243516;
      when 24 result = 0x25fed7;
      when 25 result = 0x27cd94;
when 26 result = 0x29a15b;
when 27 result = 0x2b7a3a;
when 28 result = 0x2d583f;
when 29 result = 0x2f3b79;
when 30 result = 0x3123f6;
when 31 result = 0x3311c4;
when 32 result = 0x3504f3;
when 33 result = 0x36fd92;
when 34 result = 0x38fbaf;
when 35 result = 0x3aff5b;
when 36 result = 0x3d08a4;
when 37 result = 0x3f179a;
when 38 result = 0x412c4d;
when 39 result = 0x4346cd;
when 40 result = 0x45672a;
when 41 result = 0x478d75;
when 42 result = 0x49b9be;
when 43 result = 0x4bec15;
when 44 result = 0x4e248c;
when 45 result = 0x506334;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599d16;
when 50 result = 0x5bfbb8;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 0x65907;
when 55 result = 0x68396a;
when 56 result = 0x6ac8c7;
when 57 result = 0x6d4f30;
when 58 result = 0x6fe4ba;
when 59 result = 0x728177;
when 60 result = 0x75257d;
when 61 result = 0x77cd0f;
when 62 result = 0x7a83b3;
when 63 result = 0x7d3e0c;

else // N == 64
    case index of
    when  0 result = 0x0000000000000000;
    when  1 result = 0x02C9A3E778061;
    when  2 result = 0x059B0D3158574;
    when  3 result = 0x0874518759BC8;
    when  4 result = 0x0B5586CF9890F;
    when  5 result = 0x0E3EC32D031A2;
    when  6 result = 0x11301D0125B51;
    when  7 result = 0x1429AAEA92D09;
    when  8 result = 0x172B83C7D517B;
    when  9 result = 0x1A35BE6FC875;
    when 10 result = 0x1D4873168B9AA;
    when 11 result = 0x2063B8628CD6;
    when 12 result = 0x23B7A6E756238;
    when 13 result = 0x26B4565E27C0D;
    when 14 result = 0x29E39D5F1DEEE1;
    when 15 result = 0x2DE85A6E4030B;
    when 16 result = 0x306FE0A318715;
    when 17 result = 0x33C0BB26416FF;
    when 18 result = 0x371A7373A9CB;
    when 19 result = 0x3A7DB34E59FF7;
    when 20 result = 0x3DEA64C123422;
    when 21 result = 0x4160A21F72E2A;
    when 22 result = 0x44E0860618920;
    when 23 result = 0x486A28513CD0;
    when 24 result = 0x4BFDAD5362A27;
    when 25 result = 0x4F92769D2CA7;
    when 26 result = 0x5342B569D4F82;
    when 27 result = 0x56F4736B5270A;
    when 28 result = 0x5AB07DD485429;
when 29 result = 0x5E76F15AD2148;
when 30 result = 0x6247EB03A5585;
when 31 result = 0x6623882552225;
when 32 result = 0x6A09E667F3BCD;
when 33 result = 0x71F75E8EC5F74;
when 35 result = 0x75FE564267C9;
when 36 result = 0x7A11473EB0187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x82589994CCE13;
when 39 result = 0x868D99B4492ED;
when 40 result = 0x8ACE5422AA0DB;
when 41 result = 0x8F1AE99157736;
when 42 result = 0x93737B0CDC5E5;
when 43 result = 0x97D829FDE4E50;
when 44 result = 0x9C49182A3F090;
when 45 result = 0xA0C667B5DE565;
when 46 result = 0xA550B232E255D;
when 47 result = 0xA9E6B5579FDBF;
when 48 result = 0xAE89F995AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F76F2FB5E47;
when 51 result = 0xBC1E904BC1D2;
when 52 result = 0xC1998D0B5529C;
when 53 result = 0xC67F12E57D148;
when 54 result = 0xCB720DCEF9069;
when 55 result = 0xD072D4A07897C;
when 56 result = 0xD581B0CFBA487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDFC9737B9B5F;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEA4FA2A4900A;
when 61 result = 0xEFABE615A27;
when 62 result = 0xF50765BE4548;
when 63 result = 0xFA7C1819E9008;

return result<N-1:0>;

Library pseudocode for aarch64/functions/sve/FPLogB

// FPLogB()
// ========

bits(N) FPLogB(bits(N) op, FPCRType fpcr)
assert N IN {16,32,64};

(fptype,sign,value) = FPUnpack(op, fpcr);

if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN || fptype == FPTYPE_Zero then
  FPProcessException(FPEExc_InvalidOp, fpcr);
result = -(2^(N-1));            // MinInt, 100..00
elsif fptype == FPTYPE_Infinity then
  result = 2^(N-1) - 1;           // MaxInt, 011..11
else
  // FPUnpack has already scaled a subnormal input
  value = Abs(value);
  result = 0;
  while value < 1.0 do
    value = value * 2.0;
    result = result - 1;
  while value >= 2.0 do
    value = value / 2.0;
    result = result + 1;
  FPProcessDenorm(fptype, N, fpcr);

return result<N-1:0>;

Shared Pseudocode Functions Page 3193
Library pseudocode for aarch64/functions/sve/FPMinNormal

// FPMinNormal()
// =============

bits(N) FPMinNormal(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = Zeros(E-1):'1';
frac = Zeros(F);
return sign : exp : frac;

Library pseudocode for aarch64/functions/sve/FPOne

// FPOne()
// ========

bits(N) FPOne(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-1);
frac = Zeros(F);
return sign : exp : frac;

Library pseudocode for aarch64/functions/sve/FPPointFive

// FPPointFive()
// =============

bits(N) FPPointFive(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-2):'0';
frac = Zeros(F);
return sign : exp : frac;

Library pseudocode for aarch64/functions/sve/FPProcess

// FPProcess()
// ===========

bits(N) FPProcess(bits(N) input)
assert N IN {16,32,64};
FPCRTYPE fpcr = FPCR[];
(fptype,sign,value) = FPUnpack(input, fpcr);
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
  result = FPProcessNaN(fptype, input, fpcr);
elsif fptype == FPTYPE_INFINITY then
  result = FPIInfinity(sign);
elsif fptype == FPTYPE_ZERO then
  result = FPZero(sign);
else
  result = FPRound(value, fpcr);
  FPProcessDenorm(fptype, N, fpcr);
return result;
// FPScale()
// =========

bits(N) FPScale(bits (N) op, integer scale, FPCRType fpcr)
assert N IN {16,32,64};
(fptype,sign,value) = FPUnpack(op, fpcr);
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
  result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPTYPE_Zero then
  result = FPZero(sign);
elsif fptype == FPTYPE_Infinity then
  result = FPInfinity(sign);
else
  result = FPRound(value * (2.0^scale), fpcr);
end if
FPProcessDenorm(fptype, N, fpcr);
return result;

// FPTrigMAdd()
// ============

bits(N) FPTrigMAdd(integer x, bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
assert x >= 0;
assert x < 8;
bits(N) coeff;
if op2<N-1> == '1' then
  x = x + 8;
end if
coeff = FPTrigMAddCoefficient[x];
boolean altfp = HaveAltFP() && fpcr.AH == '1';
if altfp then
  (fptype,-,-) = FPUnpack(op2, fpcr, FALSE);
  if !(fptype IN {FPTYPE_SNaN, FPTYPE_QNaN}) then
    op2<N-1> = '0';
  else
    op2<N-1> = '0';
  end if
end if
result = FPMulAdd(coeff, op1, op2, fpcr);
return result;
Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

// FPTrigMAddCoefficient()
// ================

bits(N) FPTrigMAddCoefficient[integer index]
assert N IN {16,32,64};
integer result;
if N == 16 then
  case index of
    when 0 result = 0x3c00;
    when 1 result = 0xb155;
    when 2 result = 0x2030;
    when 3 result = 0x0000;
    when 4 result = 0x0000;
    when 5 result = 0x0000;
    when 6 result = 0x0000;
    when 7 result = 0x0000;
    when 8 result = 0x3c00;
    when 9 result = 0xb800;
    when 10 result = 0x293a;
    when 11 result = 0x0000;
    when 12 result = 0x0000;
    when 13 result = 0x0000;
    when 14 result = 0x0000;
    when 15 result = 0x0000;
  elsif N == 32 then
    case index of
      when 0 result = 0x3f800000;
      when 1 result = 0xbe2aaaab;
      when 2 result = 0x3c088886;
      when 3 result = 0xb95008b9;
      when 4 result = 0x36369d6d;
      when 5 result = 0x00000000;
      when 6 result = 0x00000000;
      when 7 result = 0x00000000;
      when 8 result = 0x3f800000;
      when 9 result = 0xbf000000;
      when 10 result = 0x3d2aaaa6;
      when 11 result = 0xbab60705;
      when 12 result = 0x37cd37cc;
      when 13 result = 0x00000000;
      when 14 result = 0x00000000;
      when 15 result = 0x00000000;
  else // N == 64
    case index of
      when 0 result = 0x3ff0000000000000;
      when 1 result = 0xbfc555555555543;
      when 2 result = 0x3f811111111110f30c;
      when 3 result = 0xbf2a01a019b92fc6;
      when 4 result = 0x3ec71de351f3d22b;
      when 5 result = 0xbe5ae5e2b60f7b91;
      when 6 result = 0x3de5d8408868552f;
      when 7 result = 0x0000000000000000;
      when 8 result = 0x3ff0000000000000;
      when 9 result = 0xbe00000000000000;
      when 10 result = 0x3fa555555555536;
      when 11 result = 0xbff6c6c6c6c6c6c;
      when 12 result = 0x3efa01a019b1e8d8;
      when 13 result = 0xbe927e4f7282f468;
      when 14 result = 0x3e21ee96d2641b13;
      when 15 result = 0xbda8f76380fb401;
  return result<N-1:0>;
Library pseudocode for aarch64/functions/sve/FPTrigSMul

```c
// FPTrigSMul()
// ===========

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
result = FPMul(op1, op1, fpcr);
(fptype, sign, value) = FPUnpack(result, fpcr);
if (fptype != FPTYPE_QNaN) && (fptype != FPTYPE_SNaN) then
    result<N-1> = op2<0>;
    FPProcessDenorm(fptype, N, fpcr);
return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigSSel

```c
// FPTrigSSel()
// ============

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
assert N IN {16,32,64};
bits(N) result;
if op2<0> == '1' then
    result = FPOne(op2<1>);
else
    result = op1;
    result<N-1> = result<N-1> EOR op2<1>;
return result;
```

Library pseudocode for aarch64/functions/sve/FirstActive

```c
// FirstActive()
// ===========

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
integer elements = N DIV (esize DIV 8);
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
return '0';
```

Library pseudocode for aarch64/functions/sve/FloorPow2

```c
// FloorPow2()
// ===========

// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
assert x >= 0;
integer n = 1;
if x == 0 then return 0;
while x >= 2^n do
    n = n + 1;
return 2^(n - 1);
```

Library pseudocode for aarch64/functions/sve/HaveSVE

```c
// HaveSVE()
// =========

boolean HaveSVE()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";
```
Library pseudocode for aarch64/functions/sve/HaveSVE2

// HaveSVE2()
// =========
// Returns TRUE if the SVE2 extension is implemented, FALSE otherwise.

boolean HaveSVE2()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE2 extension";

Library pseudocode for aarch64/functions/sve/HaveSVE2AES

// HaveSVE2AES()
// =============
// Returns TRUE if the SVE2 AES extension is implemented, FALSE otherwise.

boolean HaveSVE2AES()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 AES extension";

Library pseudocode for aarch64/functions/sve/HaveSVE2BitPerm

// HaveSVE2BitPerm()
// ================
// Returns TRUE if the SVE2 Bit Permissions extension is implemented, FALSE otherwise.

boolean HaveSVE2BitPerm()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 BitPerm extension";

Library pseudocode for aarch64/functions/sve/HaveSVE2PMULL128

// HaveSVE2PMULL128()
// =================
// Returns TRUE if the SVE2 128 bit PMULL extension is implemented, FALSE otherwise.

boolean HaveSVE2PMULL128()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 128 bit PMULL extension";

Library pseudocode for aarch64/functions/sve/HaveSVE2SHA3

// HaveSVE2SHA3()
// ==============
// Returns TRUE if the SVE2 SHA3 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA3()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA3 extension";

Library pseudocode for aarch64/functions/sve/HaveSVE2SM4

// HaveSVE2SM4()
// =============
// Returns TRUE if the SVE2 SM4 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM4()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SM4 extension";

Library pseudocode for aarch64/functions/sve/HaveSVEFP32MatMulExt

// HaveSVEFP32MatMulExt()
// ======================
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP32MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP32 Matrix Multiply extension";
Library pseudocode for aarch64/functions/sve/HaveSVEFP64MatMulExt

// HaveSVEFP64MatMulExt()
// ======================
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented and FALSE otherwise.

boolean HaveSVEFP64MatMulExt()
return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP64 Matrix Multiply extension";

Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (e.g. power of two)

integer ImplementedSVEVectorLength(integer nbits)
return integer IMPLEMENTATION_DEFINED;

Library pseudocode for aarch64/functions/sve/IsEven

// IsEven()
// =======

boolean IsEven(integer val)
return val MOD 2 == 0;

Library pseudocode for aarch64/functions/sve/IsFPEnabled

// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point registers are enabled at the target exception level in the current execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
if ELUsingAArch32(el) then
  return AArch32.IsFPEnabled(el);
else
  return AArch64.IsFPEnabled(el);

Library pseudocode for aarch64/functions/sve/IsOdd

// IsOdd()
// =======

boolean IsOdd(integer val)
return val MOD 2 == 1;
// IsSVEEnabled()
// ==============
// Returns TRUE if access to SVE instructions and System registers is
// enabled at the target exception level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
if ELUsingAArch32(el) then
  return FALSE;

// Check if access disabled in CPACR_EL1
if el IN {EL0, EL1} && !IsInHost() then
  // Check SVE at EL0/EL1
  case CPACR_EL1.ZEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = el == EL0;
    when '11' disabled = FALSE;
  if disabled then return FALSE;

// Check if access disabled in CPACR_EL2
if el IN {EL0, EL1, EL2} && EL2Enabled() then
  if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
    case CPACR_EL2.ZEN of
      when 'x0' disabled = TRUE;
      when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
      when '11' disabled = FALSE;
    if disabled then return FALSE;
    else
      if CPTR_EL2.TZ == '1' then return FALSE;
  // Check if access disabled in CPTR_EL3
  if HaveEL(EL3) then
    if CPTR_EL3.EZ == '0' then return FALSE;
  return TRUE;

// LastActive()
// ============
bit LastActive(bits(N) mask, bits(N) x, integer esize)
integer elements = N DIV (esize DIV 8);
for e = elements-1 downto 0
  if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
return '0';

// LastActiveElement()
// ===================
integer LastActiveElement(bits(N) mask, integer esize)
assert esize IN {8, 16, 32, 64};
integer elements = VL DIV esize;
for e = elements-1 downto 0
  if ElemP[mask, e, esize] == '1' then return e;
return -1;

const integer MAX_PL = 256;

const integer MAX_VL = 2048;
// MaybeZeroSVEUppers()
// ====================

MaybeZeroSVEUppers(bits(2) target_el)
  boolean lower_enabled;
  if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
    return;
  if target_el == EL3 then
    if EL2Enabled() then
      lower_enabled = IsFPEnabled(EL2);
    else
      lower_enabled = IsFPEnabled(EL1);
  elsif target_el == EL2 then
    assert !ELUsingAArch32(EL2);
    if HCR_EL2.TGE == '0' then
      lower_enabled = IsFPEnabled(EL1);
    else
      lower_enabled = IsFPEnabled(EL0);
  else
    assert target_el == EL1 && !ELUsingAArch32(EL1);
    lower_enabled = IsFPEnabled(EL0);
  if lower_enabled then
    integer vl = if IsSVEEnabled(PSTATE.EL) then VL else 128;
    integer pl = vl DIV 8;
    for n = 0 to 31
      if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(_Z[n]<vl-1:0>);
      for n = 0 to 15
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
          _P[n] = ZeroExtend(_P[n]<pl-1:0>);
      if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _FFR = ZeroExtend(_FFR<pl-1:0>);
// MemNF[] - non-assignment form
// =====================================

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;
    aligned = (address == Align(address, size));
    A = SCTLR[].A;

    if !aligned && (A == '1') then
        return (bits(8*size) UNKNOWN, TRUE);

    atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, acctype, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

            for i = 1 to size-1
                (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];
                if bad then
                    return (bits(8*size) UNKNOWN, TRUE);
            else
                (value, bad) = MemSingleNF[address, size, acctype, aligned];
                if bad then
                    return (bits(8*size) UNKNOWN, TRUE);

        if BigEndian(acctype) then
            value = BigEndianReverse(value);

        return (value, FALSE);
Library pseudocode for aarch64/functions/sve/MemSingleNF

// MemSingleNF[] - non-assignment form
// ===================================
(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, AccType acctype, boolean wasaligned]
  bits(8*size) value;
  boolean iswrite = FALSE;
  AddressDescriptor memaddrdesc;

  // Implementation may suppress NF load for any reason
  if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
    return (bits(8*size) UNKNOWN, TRUE);

  // MMU or MPU
  memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

  // Non-fault load from Device memory must not be performed externally
  if memaddrdesc.memattrs.memtype == MemType_Device then
    return (bits(8*size) UNKNOWN, TRUE);

  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    return (bits(8*size) UNKNOWN, TRUE);

  // Memory array access
  if HaveTME() then
    transactional = TSTATE.depth > 0;
    accdesc = CreateAccessDescriptor(acctype, transactional);
  else
    accdesc = CreateAccessDescriptor(acctype);
  if HaveMTEExt() then
    if AArch64.AccessIsTagChecked(address, acctype) then
      bits(4) ptag = AArch64.PhysicalTag(address);
      if !AArch64.CheckTag(memaddrdesc, ptag, iswrite) then
        return (bits(8*size) UNKNOWN, TRUE);
    value = _Mem[memaddrdesc, size, accdesc, iswrite];
  return (value, FALSE);

Library pseudocode for aarch64/functions/sve/NoneActive

// NoneActive()
// ============
bit NoneActive(bits(N) mask, bits(N) x, integer esize)
  integer elements = N DIV (esize DIV 8);
  for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
  return '1';
Library pseudocode for aarch64/functions/sve/P

// P[] - non-assignment form
// =========================

bits(width) P[integer n]
    assert n >= 0 && n <= 31;
    assert width == PL;
    return P[n]<width-1:0>;

// P[] - assignment form
// =====================

P[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == PL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        P[n] = ZeroExtend(value);
    else
        P[n]<width-1:0> = value;

Library pseudocode for aarch64/functions/sve/PL

// PL - non-assignment form
// ========================

integer PL
    return VL DIV 8;

Library pseudocode for aarch64/functions/sve/PredTest

// PredTest()
// =========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;

Library pseudocode for aarch64/functions/sve/ReducePredicated

// ReducePredicated()
// ==================

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;
    for e = 0 to elements-1
        if e * esize < N && ElemP[mask, e, esize] == '1' then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;
    return Reduce(op, operand, esize);
Library pseudocode for aarch64/functions/sve/Reverse

```c
// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    bits(N) result;
    integer sw = N DIV M;
    assert N == sw * M;
    for s = 0 to sw-1
        Elem[result, sw - 1 - s, M] = Elem[word, s, M];
    return result;
```

Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```c
// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';
    exception = ExceptionSyndrome(Exception_SVEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    if route to e2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/sve/SVECmp

```c
enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```
Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)

   bits(64) imm;
   (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE);

   // Check for 8 bit immediates
   if !IsZero(imm<7:0>) then
      // Check for 'ffffffffffffffffxy' or '00000000000000xy'
      if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
         return FALSE;
      // Check for 'ffxyffxyffxyffxy' or '00000000000000xyxy'
      if !IsZero(imm<63:32>) && !IsOnes(imm<63:32>) then
         return FALSE;
      // Check for 'xyxyxyxyxyxyxyxyxyxyxyy'
      if !IsZero(imm<63:16>) && !IsOnes(imm<63:16>) then
         return FALSE;
   // Check for 16 bit immediates
   else
      // Check for 'ffffffffffffffxy00' or '000000000000xy00'
      if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
         return FALSE;
      // Check for 'ffxy00ffxy00ffxy00' or '0000y000000xy00y00'
      if !IsZero(imm<63:32>) && !IsOnes(imm<63:32>) then
         return FALSE;
      // Check for 'xy00xy00xy00xy00y00'
      if !IsZero(imm<63:16>) && !IsOnes(imm<63:16>) then
         return FALSE;
   return TRUE;

Library pseudocode for aarch64/functions/sve/ShiftSat

// ShiftSat()
// =========

integer ShiftSat(integer shift, integer esize)

   if shift > esize+1 then return esize+1;
   elsif shift < -(esize+1) then return -(esize+1);
   return shift;

Library pseudocode for aarch64/functions/sve/System

array bits(MAX_VL) _Z[0..31];
array bits(MAX_PL) _P[0..15];
bits(MAX_PL) _FFR;

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/sve/VL

// VL - non-assignment form
// ------------------------

integer VL
    integer vl;
    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UINT(ZCR_EL1.LEN);
    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UINT(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UINT(ZCR_EL2.LEN));
    if PSTATE.EL == EL3 then
        vl = UINT(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UINT(ZCR_EL3.LEN));
    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);
    return vl;

Library pseudocode for aarch64/functions/sve/Z

// Z[] - non-assignment form
// -------------------------

bits(width) Z[integer n]
    assert n >= 0 && n <= 31;
    assert width == VL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// ----------------------

Z[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == VL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value);
    else
        _Z[n]<width-1:0> = value;

Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

// CNTKCTL[] - non-assignment form
// -------------------------------

CNTKCTLType CNTKCTL[]
    bits(64) r;
    if IsInHost() then
        r = CNTKCTL_EL2;
        return r;
    r = CNTKCTL_EL1;
    return r;

Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

type CNTKCTLType;
Library pseudocode for aarch64/functions/sysregisters/CPACR

// CPACR[] - non-assignment form
// ===============

CPACRTyp

bits(64) r;
if IsInHost() then
  r = CPTR_EL2;
  return r;
  r = CPACR_EL1;
  return r;

Library pseudocode for aarch64/functions/sysregisters/CPACRTyp
type CPACRTyp;

Library pseudocode for aarch64/functions/sysregisters/ELR

// ELR[] - non-assignment form
// ===============

bits(64) ELR[bits(2) el]
bits(64) r;
case el of
  when EL1 r = ELR_EL1;
  when EL2 r = ELR_EL2;
  when EL3 r = ELR_EL3;
  otherwise Unreachable();
  return r;

// ELR[] - non-assignment form
// ===============

bits(64) ELR[]
assert PSTATE.EL != EL0;
return ELR[PSTATE.EL];

// ELR[] - assignment form
// ===============

ELR[bits(2) el] = bits(64) value
bits(64) r = value;
case el of
  when EL1 ELR_EL1 = r;
  when EL2 ELR_EL2 = r;
  when EL3 ELR_EL3 = r;
  otherwise Unreachable();
  return;

// ELR[] - assignment form
// ===============

ELR[] = bits(64) value
assert PSTATE.EL != EL0;
ELR[PSTATE.EL] = value;
return;
```plaintext
// ESR[] - non-assignment form
// ===========================
ESRType ESR[bits(2) regime] = bits(64) r;
case regime of
  when EL1  r = ESR_EL1;
  when EL2  r = ESR_EL2;
  when EL3  r = ESR_EL3;
  otherwise unreachable();
return r;

// ESR[] - non-assignment form
// ===========================
ESRType ESR[] = ESR[SITranslationRegime()];

// ESR[] - assignment form
// =======================
ESR[bits(2) regime] = ESRType value = bits(64) r = value;
case regime of
  when EL1  ESR_EL1 = r;
  when EL2  ESR_EL2 = r;
  when EL3  ESR_EL3 = r;
  otherwise unreachable();
return;

// ESR[] - assignment form
// =======================
ESR[] = ESRType value = ESR[SITranslationRegime()] = value;
```

**Library pseudocode for aarch64/functions/sysregisters/ESR**

type ESRType;

**Library pseudocode for aarch64/functions/sysregisters/ESRType**
Library pseudocode for aarch64/functions/sysregisters/FAR

// FAR[] - non-assignment form
// --------------------------

bits(64) FAR[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1 r = FAR_EL1;
    when EL2 r = FAR_EL2;
    when EL3 r = FAR_EL3;
    otherwise Unreachable();
  return r;

// FAR[] - non-assignment form
// --------------------------

bits(64) FAR[]
  return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// ------------------------

FAR[bits(2) regime] = bits(64) value
  bits(64) r = value;
  case regime of
    when EL1 FAR_EL1 = r;
    when EL2 FAR_EL2 = r;
    when EL3 FAR_EL3 = r;
    otherwise Unreachable();
  return;

// FAR[] - assignment form
// ------------------------

FAR[] = bits(64) value
  FAR[S1TranslationRegime()] = value;
  return;

Library pseudocode for aarch64/functions/sysregisters/MAIR

// MAIR[] - non-assignment form
// ---------------------------

MAIRType MAIR[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1 r = MAIR_EL1;
    when EL2 r = MAIR_EL2;
    when EL3 r = MAIR_EL3;
    otherwise Unreachable();
  return r;

// MAIR[] - non-assignment form
// ---------------------------

MAIR[] = bits(64) value
  MAIR[S1TranslationRegime()] = value;
  return;

// MAIR[] - assignment form
// --------------------------

Library pseudocode for aarch64/functions/sysregisters/MAIRTType

type MAIRTType;
Library pseudocode for aarch64/functions/sysregisters/SCTLR

// SCTLR[] - non-assignment form
// =============================
SCTLRType SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// =============================
SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];

Library pseudocode for aarch64/functions/sysregisters/SCTLRType

type SCTLRType;

Library pseudocode for aarch64/functions/sysregisters/VBAR

// VBAR[] - non-assignment form
// =============================
bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// =============================
bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

// AArch64.AllocationTagAccessIsEnabled()
// ==================================================================
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(AccType acctype)
    bits(2) el = AArch64.AccessUsesEL(acctype);
    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && el == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && el == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && el == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && el == EL0 && !EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && el == EL0 && !EL2Enabled() then
        return FALSE;
    else
        return TRUE;

Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

// AArch64.CheckSystemAccess()
// ===========================
AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
    bits(4) crm, bits(3) op2, bits(5) rt, bit read)
    if (TSTATE.depth > 0 &&
        !CheckTransactionalSystemAccess(op0, op1, crn, crm, op2, read)) then
        FailTransaction(TMFailure_ERR, FALSE);
    return;

Library pseudocode for aarch64/functions/system/AArch64.ChooseNonExcludedTag

// AArch64.ChooseNonExcludedTag()
// ==============================
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag, bits(4) offset, bits(16) exclude)
    if IsOnes(exclude) then
        return '0000';
    if offset == '0000' then
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';
        while offset != '0000' do
            offset = offset - '0001';
            tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';
        return tag;
Library pseudocode for aarch64/functions/system/AArch64.ExecutingATS1xPInstr

// AArch64.ExecutingATS1xPInstr()
// ================================================
// Return TRUE if current instruction is AT S1E1R/WP

boolean AArch64.ExecutingATS1xPInstr()
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<22+:10> == '1101010100' then
        op1 = instr<16+:3>;
        CRn = instr<12+:4>;
        CRm = instr<8+:4>;
        op2 = instr<5+:3>;
        return op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'};
    else
        return FALSE;

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr

// AArch64.ExecutingBROrBLROrRetInstr()
// ====================================
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBROrBLROrRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

// AArch64.ExecutingBTIInstr()
// ===========================
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm = instr<11:8>;
        op2 = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;

Library pseudocode for aarch64/functions/system/AArch64.ExecutingERETInstr

// AArch64.ExecutingERETInstr()
// ============================
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110101111110000';
Library pseudocode for aarch64/functions/system/AArch64.NextRandomTagBit

```c
// AArch64.NextRandomTagBit()
// =========================
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    bits(16) lfsr = RGSR_EL1.SEED;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSR_EL1.SEED = top:lfsr<15:1>;
    return top;
```

Library pseudocode for aarch64/functions/system/AArch64.RandomTag

```c
// AArch64.RandomTag()
// ===============
// Generate a random Allocation Tag.

bits(4) AArch64.RandomTag()
    bits(4) tag;
    for i = 0 to 3
        tag<i> = AArch64.NextRandomTagBit();
    return tag;
```

Library pseudocode for aarch64/functions/system/AArch64.SysInstr

```c
// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

Library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResult

```c
// Execute a system instruction with read (result operand).
// Returns the result of the instruction.
bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);
```

Library pseudocode for aarch64/functions/system/AArch64.SysRegRead

```c
// Read from a system register and return the contents of the register.
bits(64) AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2);
```

Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

```c
// Write to a system register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

Library pseudocode for aarch64/functions/system/BTypeCompatible

```c
boolean BTypeCompatible;
```
Library pseudocode for aarch64/functions/system/BTypeCompatible_BTI

// BTypeCompatible_BTI
// ------------------
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
  case hintcode of
    when '00'
      return FALSE;
    when '01'
      return PSTATE.BTYPE != '11';
    when '10'
      return PSTATE.BTYPE != '10';
    when '11'
      return TRUE;
  endcase;

Library pseudocode for aarch64/functions/system/BTypeCompatible_PACIXSP

// BTypeCompatible_PACIXSP()
// ------------------------
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
  if PSTATE.BTYPE IN {'01', '10'} then
    return TRUE;
  elsif PSTATE.BTYPE == '11' then
    index = if PSTATE.EL == EL0 then 35 else 36;
    return SCTLR[]<index> == '0';
  else
    return FALSE;
  endcase;

Library pseudocode for aarch64/functions/system/BTypeNext

bits(2) BTypeNext;

Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTag

// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit(). Implementations
// may choose to behave the same as GCR_EL1.RRND=0.

bits(4) ChooseRandomNonExcludedTag(bits(16) exclude);

Library pseudocode for aarch64/functions/system/InGuardedPage

boolean InGuardedPage;

Library pseudocode for aarch64/functions/system/SetBTypeCompatible

// SetBTypeCompatible()
// -------------------
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
  BTypeCompatible = x;
Library pseudocode for aarch64/functions/system/SetBTypeNext

// SetBTypeNext()
// ==============
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;

Library pseudocode for aarch64/functions/tme/CheckTMEEnabled

// CheckTMEEnabled()
// =================
// Returns TRUE if access to TME instruction is enabled, FALSE otherwise.

CheckTMEEnabled()
    if PSTATE.EL IN {EL0, EL1, EL2} && HaveEL(EL3) then
        if SCR_EL3.TME == '0' then UNDEFINED;
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
            if HCR_EL2.TME == '0' then UNDEFINED;
    return;

Library pseudocode for aarch64/functions/tme/CheckTransactionalSystemAccess

// CheckTransactionalSystemAccess()
// ================================
// Returns TRUE if an AArch64 MSR, MRS, or SYS instruction is permitted in
// Transactional state, based on the opcode's encoding, and FALSE otherwise.

boolean CheckTransactionalSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bit read)
    case read:op0:op1:crn:crm:op2 of
        when '0 00 011 011 0000 001' return TRUE;  // MSR (imm): DAIFSet, DAIFClr
        when '0 01 011 011 0100 001' return TRUE;  // DC ZVA
        when '0 11 011 011 0000 001' return TRUE;  // MSR: NZCV, DAIF
        when '0 11 011 011 0100 001' return TRUE;  // MSR: FPCR, FPSR
        when '0 11 011 1001 1100 100' return TRUE; // MRS: PMSWINC_EL0
        when '1 11 xxx 000 0100 000' return TRUE;  // MSR: ICC_PMR_EL1
        when '1 11 xxx 1001 1100 100' return TRUE; // MRS: ICC_HPPIRx_EL1
        when '1 11 xxx 1101 1100 100' return TRUE; // MRS: ICC_RPR_EL1
        when '1 11 xxx 1110 1100 100' return TRUE; // MRS: ICC_RPR_EL1
        when '0 01 011 011 0100 001' return TRUE;  // CFP RCTX
        when '0 01 011 011 0110 011' return TRUE;  // CPP RCTX
        when 'x 11 xxx 1110 xxxx xxxx' return boolean IMPLEMENTATION_DEFINED; // MRS: op1=3, Cn=11,15
        otherwise return FALSE;  // all other SYS, SYSL, MRS, MSR

Library pseudocode for aarch64/functions/tme/CommitTransactionalWrites

// Makes all transactional writes to memory observable by other PEs and reset
// the transactional read and write sets.
CommitTransactionalWrites();

Library pseudocode for aarch64/functions/tme/DiscardTransactionalWrites

// Discards all transactional writes to memory and reset the transactional
// read and write sets.
DiscardTransactionalWrites();
Library pseudocode for aarch64/functions/tme/FailTransaction

// FailTransaction()
// ================

FailTransaction(TMFailure cause, boolean retry)
    FailTransaction(cause, retry, FALSE, Zeros(15));
    return;

// FailTransaction()
// ================
// Exits Transactional state and discards transactional updates to registers
// and memory.

FailTransaction(TMFailure cause, boolean retry, boolean interrupt, bits(15) reason)
    assert !retry || !interrupt;
    DiscardTransactionalWrites();
    if cause != TMFailure_TRIVIAL then // For trivial implementation no transaction checkpoint was taken
        RestoreTransactionCheckpoint();
    ClearExclusiveLocal(ProcessorID());
    bits(64) result = Zeros();
    result<23> = if interrupt then '1' else '0';
    result<15> = if retry && !interrupt then '1' else '0';
    case cause of
        when TMFailure_TRIVIAL result<24> = '1';
        when TMFailure_DBG result<22> = '1';
        when TMFailure_NEST result<21> = '1';
        when TMFailure_SIZE result<20> = '1';
        when TMFailure_ERR result<19> = '1';
        when TMFailure_IMP result<18> = '1';
        when TMFailure_MEM result<17> = '1';
        when TMFailure_CNCL result<16> = '1'; result<14:0> = reason;
    TSTATE.depth = 0;
    X[TSTATE.Rt] = result;
    BranchTo(TSTATE.nPC, BranchType_TMFAIL);
    EndOfInstruction();
    return;
Library pseudocode for aarch64/functions/tme/RestoreTransactionCheckpoint

// RestoreTransactionCheckpoint()
//=============================
// Restores part of the PE registers from the transaction checkpoint.

RestoreTransactionCheckpoint()
  SP[] = TSTATE.SP;
  ICC_PMR_EL1 = TSTATE.ICC_PMR_EL1;
  PSTATE.<N,Z,C,V> = TSTATE.nzcv;
  PSTATE.<D,A,I,F> = TSTATE.<D,A,I,F>;
  for n = 0 to 30
    X[n] = TSTATE.X[n];
  if IsFPEnabled(PSTATE.EL) then
    if IsSVEEnabled(PSTATE.EL) then
      for n = 0 to 31
        Z[n] = TSTATE.Z[n]<VL-1:0>;
      for n = 0 to 15
        P[n] = TSTATE.P[n]<PL-1:0>;
        FFR[] = TSTATE.FFR<PL-1:0>;
    else
      for n = 0 to 31
        V[n] = TSTATE.Z[n]<127:0>;
    end:
  end:
  FPCR = TSTATE.FPCR;
  FPSR = TSTATE.FPSR;
  return;

Library pseudocode for aarch64/functions/tme/StartTrackingTransactionalReadsWrites

// Starts tracking transactional reads and writes to memory.
StartTrackingTransactionalReadsWrites();

Library pseudocode for aarch64/functions/tme/TMFailure

enumeration TMFailure {
  TMFailure_CNCL,    // Executed a TCANCEL instruction
  TMFailure_DBG,     // A debug event was generated
  TMFailure_ERR,     // A non-permissible operation was attempted
  TMFailure_NEST,    // The maximum transactional nesting level was exceeded
  TMFailure_SIZE,    // The transactional read or write set limit was exceeded
  TMFailure_MEM,     // A transactional conflict occurred
  TMFailure_TRIVIAL, // Only a TRIVIAL version of TM is available
  TMFailure_IMP      // Any other failure cause
};

Library pseudocode for aarch64/functions/tme/TMState

type TMState is (  
  integer       depth,       // Transaction nesting depth
  integer       Rt,          // TSTART destination register
  bits(64)      nPC,         // Fallback instruction address
  array[0..30] of bits(64)  X,  // General purpose registers
  array[0..31] of bits(MAX_VL) Z, // Vector registers
  array[0..15] of bits(MAX_PL) P, // Predicate registers
  bits(MAX_PL)  FFR,         // First Fault Register
  bits(64)      SP,          // Stack Pointer at current EL
  bits(64)      FPCR,        // Floating-point Control Register
  bits(64)      FPSR,        // Floating-point Status Register
  bits(64)      ICC_PMR_EL1, // Interrupt Controller Interrupt Priority Mask Register
  bits(4)       nzcv,        // Condition flags
  bits(1)       D,           // Debug mask bit
  bits(1)       A,           // SError interrupt mask bit
  bits(1)       I,           // IRQ mask bit
  bits(1)       F            // FIQ mask bit
)
Library pseudocode for aarch64/functions/tme/TSTATE

TMState TSTATE;

Library pseudocode for aarch64/functions/tme/TakeTransactionCheckpoint

// TakeTransactionCheckpoint()
// ===========================
// Captures part of the PE registers into the transaction checkpoint.

TakeTransactionCheckpoint()
  TSTATE.SP = SP[];
  TSTATE.ICC_PMR_EL1 = ICC_PMR_EL1;
  TSTATE.nzcv = PSTATE.<N,Z,C,V>;
  TSTATE.<D,A,I,F> = PSTATE.<D,A,I,F>;
  for n = 0 to 30
    TSTATE.X[n] = X[n];
  if IsFPEnabled(PSTATE.EL) then
    if IsSVEEnabled(PSTATE.EL) then
      for n = 0 to 31
        TSTATE.Z[n]<VL-1:0> = Z[n];
      for n = 0 to 15
        TSTATE.P[n]<PL-1:0> = P[n];
        TSTATE.FFR<PL-1:0> = FFR[];
      else
        for n = 0 to 31
          TSTATE.Z[n]<127:0> = V[n];
      TSTATE.FPCR = FPCR;
      TSTATE.FPSR = FPSR;
  return;

Library pseudocode for aarch64/functions/tme/TransactionStartTrap

// TransactionStartTrap()
// ========================
// Traps the execution of TSTART instruction.

TransactionStartTrap(integer dreg)
  bits(64) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x0;
  exception = ExceptionSyndrome(Exception_TSTARTAccessTrap);
  exception.syndrome<9:5> = dreg<4:0>;
  if UInt(PSTATE.EL) > UInt(EL1) then
    targetEL = PSTATE.EL;
  elsif EL2Enabled() && HCR_EL2.TGE == '1' then
    targetEL = EL2;
  else
    targetEL = EL1;
  AArch64.TakeException(targetEL, exception, preferred_exception_return, vect_offset);
Library pseudocode for aarch64/instrs/branch/eret/AArch64.ExceptionReturn

// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc, bits(64) spsr)

    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);
    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAAArch32() then
        // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                 // T32
        else
            new_pc<1:0> = '00';              // A32
        else
            // Return to AArch64
            // ELR_ELx[63:56] might include a tag
            new_pc = AArch64.BranchAddr(new_pc);  
            BranchExceptionReturnRecord(new_pc, source_el);
    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType_ERET);
    else
        BranchToAddr(new_pc, BranchType_ERET);

Library pseudocode for aarch64/instrs/countop/CountOp

enumeration CountOp    {
    CountOp_CLZ, CountOp_CLS, CountOp_CNT;
}

Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend

// DecodeRegExtend()
// ===============
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
    when '000' return ExtendType_UXTB;
    when '001' return ExtendType_UXTH;
    when '010' return ExtendType_UXTW;
    when '011' return ExtendType_UXTX;
    when '100' return ExtendType_SXTB;
    when '101' return ExtendType_SXTH;
    when '110' return ExtendType_SXTW;
    when '111' return ExtendType_SXTX;
Library pseudocode for aarch64/instrs/extendreg/ExtendReg

// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift)
  assert shift >= 0 & shift <= 4;
  bits(N) val = X[reg];
  boolean unsigned;
  integer len;

  case exttype of
    when ExtendType_SXTB unsigned = FALSE; len = 8;
    when ExtendType_SXTH unsigned = FALSE; len = 16;
    when ExtendType_SXTW unsigned = FALSE; len = 32;
    when ExtendType_SXTX unsigned = FALSE; len = 64;
    when ExtendType_UXTB unsigned = TRUE; len = 8;
    when ExtendType_UXTH unsigned = TRUE; len = 16;
    when ExtendType_UXTW unsigned = TRUE; len = 32;
    when ExtendType_UXTX unsigned = TRUE; len = 64;

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);

Library pseudocode for aarch64/instrs/extendreg/ExtendType

enumeration ExtendType  {
  ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
  ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX;
}

Library pseudocode for aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

enumeration FPMaxMinOp  {
  FPMaxMinOp_MAX, FPMaxMinOp_MIN,
  FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM;
}

Library pseudocode for aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

enumeration FPUnaryOp   {
  FPUnaryOp_ABS, FPUnaryOp_MOV,
  FPUnaryOp_NEG, FPUnaryOp_SQRT;
}

Library pseudocode for aarch64/instrs/float/convert/fpconvop/FPConvOp

enumeration FPConvOp     {
  FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
  FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF,
  FPConvOp_CVT_FtoI_JS
};
/ BFXPreferred()
// ==============
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
// DecodeBitMasks()
// ===============
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

// (bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
// bits(64) tmask, wmask;
// bits(6) tmask_and, wmask_and;
// bits(6) tmask_or, wmask_or;
// bits(6) levels;

// Compute log2 of element size
// 2^len must be in range [2, M]
len = HighestSetBit(immN:NOT(imms));
if len < 1 then UNDEFINED;
assert M >= (1 << len);

// Determine S, R and S - R parameters
levels = ZeroExtend(Ones(len), 6);

// For logical immediates an all-ones value of S is reserved
// since it would generate a useless all-ones result (many times)
if immediate && (imms AND levels) == levels then
  UNDEFINED;
S = UInt(imms AND levels);
R = UInt(immr AND levels);
diff = S - R;  // 6-bit subtract with borrow

// From a software perspective, the remaining code is equivalent to:
//   esize = 1 << len;
//   d = UInt(diff<len-1:0>);
//   welem = ZeroExtend(Ones(S + 1), esize);
//   telem = ZeroExtend(Ones(d + 1), esize);
//   wmask = Replicate(ROR(welem, R));
//   tmask = Replicate(telem);
//   return (wmask, tmask);

// Compute "top mask"
tmask_and = diff<5:0> OR NOT(levels);
tmask_or  = diff<5:0> AND levels;

tmask = Ones(64);
tmask = ((tmask
  AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
  OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));

// optimization of first step:
// tmask = Replicate(tmask_and<0> : '1', 32);
tmask = ((tmask
  AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
  OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
tmask = ((tmask
  AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
  OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
tmask = ((tmask
  AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
  OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
tmask = ((tmask
  AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
  OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
tmask = ((tmask
  AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
  OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

// Compute "wraparound mask"
wmask_and = immr OR NOT(levels);
wmask_or  = immr AND levels;

wmask = Zeros(64);
wmask = ((wmask
AND \texttt{Replicate(Ones}(1) : \texttt{Replicate}(\texttt{wmask}\_and<0>, 1), 32)) \\
OR \texttt{Replicate(Replicate}(\texttt{wmask}\_or<0>, 1) : \texttt{Zeros}(1), 32)); \\

// optimization of first step: \\
// wmask = \texttt{Replicate(}\texttt{wmask}\_or<0> : '0', 32); \\
wmask = ((\texttt{wmask} \\
AND \texttt{Replicate(Ones}(2) : \texttt{Replicate}(\texttt{wmask}\_and<1>, 2), 16)) \\
OR \texttt{Replicate(Replicate}(\texttt{wmask}\_or<1>, 2) : \texttt{Zeros}(2), 16)); \\
wmask = ((\texttt{wmask} \\
AND \texttt{Replicate(Ones}(4) : \texttt{Replicate}(\texttt{wmask}\_and<2>, 4), 8)) \\
OR \texttt{Replicate(Replicate}(\texttt{wmask}\_or<2>, 4) : \texttt{Zeros}(4), 8)); \\
wmask = ((\texttt{wmask} \\
AND \texttt{Replicate(Ones}(8) : \texttt{Replicate}(\texttt{wmask}\_and<3>, 8), 4)) \\
OR \texttt{Replicate(Replicate}(\texttt{wmask}\_or<3>, 8) : \texttt{Zeros}(8), 4)); \\
wmask = ((\texttt{wmask} \\
AND \texttt{Replicate(Ones}(16) : \texttt{Replicate}(\texttt{wmask}\_and<4>, 16), 2)) \\
OR \texttt{Replicate(Replicate}(\texttt{wmask}\_or<4>, 16) : \texttt{Zeros}(16), 2)); \\
wmask = ((\texttt{wmask} \\
AND \texttt{Replicate(Ones}(32) : \texttt{Replicate}(\texttt{wmask}\_and<5>, 32), 1)) \\
OR \texttt{Replicate(Replicate}(\texttt{wmask}\_or<5>, 32) : \texttt{Zeros}(32), 1)); \\

if \texttt{diff<6>} != '0' then // borrow from S - R \\
wmask = \texttt{wmask AND tmask}; \\
else \\
wmask = \texttt{wmask OR tmask}; \\

return (\texttt{wmask<}\texttt{M-1:0>}, \texttt{tmask<}\texttt{M-1:0>});

---

**Library pseudocode for aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp**


---

**Library pseudocode for aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred**

// MoveWidePreferred() 
// ================ 
// 
// Return TRUE if a bitmask immediate encoding would generate an immediate 
// value that could also be represented by a single MOVZ or MOVN instruction. 
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr) 
    integer S = 
        \texttt{UInt}(imms); 
    integer R = 
        \texttt{UInt}(immr); 
    integer width = if sf == '1' then 64 else 32; 

    // element size must equal total immediate size 
    if sf == '1' & immN:imms != '1xxxxxx' then 
        return FALSE; 
    if sf == '0' & immN:imms != '00xxxxx' then 
        return FALSE; 

    // for MOVZ must contain no more than 16 ones 
    if S <= 16 then 
        // ones must not span halfword boundary when rotated 
        return (-R MOD 16) <= (15 - S); 
    
    // for MOVN must contain no more than 16 zeros 
    if S >= width - 15 then 
        // zeros must not span halfword boundary when rotated 
        return (R MOD 16) <= (S - (width - 15)); 

    return FALSE;
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00' return ShiftType_LSL;
        when '01' return ShiftType_LSR;
        when '10' return ShiftType_ASR;
        when '11' return ShiftType_ROR;

// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount)
    bits(N) result = X[reg];
    case shiftype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;

// ShiftType
// -----------

enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};

// LogicalOp
// ----------

enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

// MemAtomicOp
// -------------


// MemOp
// -------

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
Prefetch(bit(64) address, bit(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;  // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;  // PLI: preload instructions
        when '10' hint = Prefetch_WRITE; // PST: prepare for store
        when '11' return;               // unallocated hint
        target = UInt(prfop<2:1>);      // target cache level
        stream = (prfop<0> != '0');     // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;

MemBarrierOp enumeration { MemBarrierOp_DSB         // Data Synchronization Barrier
                          , MemBarrierOp_DMB         // Data Memory Barrier
                          , MemBarrierOp_ISB         // Instruction Synchronization Barrier
                          , MemBarrierOp_SSBB        // Speculative Synchronization Barrier to VA
                          , MemBarrierOp_PSSBB       // Speculative Synchronization Barrier to PA
                          , MemBarrierOp_SB          // Speculation Barrier
                          };

SystemHintOp enumeration { SystemHintOp_NOP,
                          SystemHintOp_YIELD,
                          SystemHintOp_WFE,
                          SystemHintOp_WFI,
                          SystemHintOp_SEV,
                          SystemHintOp_SEVL,
                          SystemHintOp_DGH,
                          SystemHintOp_ESB,
                          SystemHintOp_PSB,
                          SystemHintOp_TSB,
                          SystemHintOp_BTI,
                          SystemHintOp_WFET,
                          SystemHintOp_WFIT,
                          SystemHintOp_CSDB
                          };

PSTATEField enumeration { PSTATEField_DAIFSet,
                          PSTATEField_DAIFClr,
                          PSTATEField_PAN, // Armv8.1
                          PSTATEField_UA0, // Armv8.2
                          PSTATEField_DIT, // Armv8.4
                          PSTATEField_SSB5,
                          PSTATEField_TCO, // Armv8.5
                          PSTATEField_CSR,
                          PSTATEField_SP
                          };

Library pseudocode for aarch64/instrs/memory/prefetch/Prefetch

Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

Library pseudocode for aarch64/instrs/system/hints/syshintop/SystemHintOp

Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

Shared Pseudocode Functions
Library pseudocode for aarch64/instrs/system/sysops/sysop/SysOp

// SysOp()
// ========

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2) {
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT; // S1E1R
        when '100 0111 1000 000' return Sys_AT; // S1E2R
        when '110 0111 1000 000' return Sys_AT; // S1E3R
        when '000 0111 1000 001' return Sys_AT; // S1E1W
        when '100 0111 1000 001' return Sys_AT; // S1E2W
        when '110 0111 1000 001' return Sys_AT; // S1E3W
        when '000 0111 1000 010' return Sys_AT; // S1E0R
        when '000 0111 1000 011' return Sys_AT; // S1E0W
        when '100 0111 1000 100' return Sys_AT; // S12E1R
        when '100 0111 1000 101' return Sys_AT; // S12E1W
        when '100 0111 1000 110' return Sys_AT; // S12E0R
        when '100 0111 1000 111' return Sys_AT; // S12E0W
        when '011 0111 0100 001' return Sys_DC; // ZVA
        when '000 0111 0110 001' return Sys_DC; // IVAC
        when '000 0111 0110 010' return Sys_DC; // ISW
        when '011 0111 1010 001' return Sys_DC; // CVAC
        when '000 0111 1010 010' return Sys_DC; // CSW
        when '011 0111 1011 001' return Sys_DC; // CVAU
        when '011 0111 1110 001' return Sys_DC; // CVADP
        when '000 0111 1011 001' return Sys_DC; // CSW
        when '011 0111 1110 001' return Sys_DC; // CIVAC
        when '011 0111 1110 100' return Sys_DC; // CIVD
        when '011 0111 1110 101' return Sys_DC; // CVAU
        when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
        when '100 1000 0011 000' return Sys_TLBI; // ALLE2
        when '110 1000 0011 000' return Sys_TLBI; // ALLE3
        when '000 1000 0011 001' return Sys_TLBI; // VAE1
        when '100 1000 0011 001' return Sys_TLBI; // VAE2
        when '110 1000 0011 001' return Sys_TLBI; // VAE3
        when '000 1000 0011 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0011 011' return Sys_TLBI; // VAAE1
        when '100 1000 0011 100' return Sys_TLBI; // VALE1
        when '100 1000 0011 101' return Sys_TLBI; // VALE2
        when '110 1000 0011 101' return Sys_TLBI; // VALE3
        when '000 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
        when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
        when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
        when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
        when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 000' return Sys_TLBI; // ALLE2
        when '110 1000 0111 000' return Sys_TLBI; // ALLE3
        when '000 1000 0111 001' return Sys_TLBI; // VAE1
        when '100 1000 0111 001' return Sys_TLBI; // VAE2
        when '110 1000 0111 001' return Sys_TLBI; // VAE3
        when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI; // VAAE1
        when '100 1000 0111 100' return Sys_TLBI; // VALE1
        when '100 1000 0111 101' return Sys_TLBI; // VALE2
        when '110 1000 0111 101' return Sys_TLBI; // VALE3
        when '000 1000 0111 110' return Sys_TLBI; // VMALLS12E1
        when '000 1000 0111 111' return Sys_TLBI; // VAALE1
    return Sys_SYS;
}

Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
constant bits(16) ASID_NONE = Zeros();

// Broadcast
// =========
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain.
Broadcast(Shareability shareability, TLBIRecord r)
    IMPLEMENTATION_DEFINED;

// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.
boolean HasLargeAddress(Regime regime)
    if !Have52BitIPAAndPASpaceExt() then
        return FALSE;
    case regime of
        when Regime_EL3
            return TCR_EL3<32> == '1';
        when Regime_EL2
            return TCR_EL2<32> == '1';
        when Regime_EL10
            return TCR_EL2<59> == '1';
        when Regime_EL10
            return TCR_EL1<59> == '1';
        otherwise
            Unreachable();
};

// Library pseudocode for aarch64/instrs/system/sysops/tlbi/Regime

enumeration Regime {
    Regime_EL10,       // EL1&0
    Regime_EL20,       // EL2&0
    Regime_EL2,        // EL2
    Regime_EL3         // EL3
};

// Library pseudocode for aarch64/instrs/system/sysops/tlbi/SecurityState

enumeration SecurityState {
    SS_NonSecure,
    SS_Secure
};
// SecurityStateAtEL()
// ===================
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
if !HaveEL(EL3) then
    if boolean IMPLEMENTATION_DEFINED "Secure-only implementation" then
        return SS_Secure;
    else
        return SS_NonSecure;
elsif EL == EL3 then
    return SS_Secure;
else
    // For EL call only when EL2 is enabled in current security state
    assert(EL != EL2 || EL2Enabled());
    if !ELUsingAArch32(EL3) then
        return if SCR_EL3.NS == '1' then SS_NonSecure else SS_Secure;
    else
        return if SCR.NS == '1' then SS_NonSecure else SS_Secure;
end if

Library pseudocode for aarch64/instrs/system/sysops/tlbi/Shareability

enumeration Shareability {
    Shareability_None,
    Shareability_Inner,
    Shareability_Outer
};

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI

// TLBI
// =====
// IMPLEMENTATION DEFINED TLBI function.

TLBI(TLBIRecord r)
IMPLEMENTATION_DEFINED;

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBILevel

enumeration TLBILevel {
    TLBILevel.Any,
    TLBILevel.Last
};

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIOp

enumeration TLBIOp {
    TLBIOp.ALL,
    TLBIOp.ASID,
    TLBIOp.IPAS2,
    TLBIOp.VAA,
    TLBIOp.VA,
    TLBIOp.VMALL,
    TLBIOp.VMALLS12,
    TLBIOp.RIPAS2,
    TLBIOp.RVAA,
    TLBIOp.RVA,
};
Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRecord

type TLBIRecord is (  
    TLBIOp         op,  
    SecurityState security,  
    Regime        regime,  
    bits(16) vmid,  
    bits(16) asid,  
    TLBILevel     level,  
    TLBI MemAttr attr,  
    FullAddress address,  // VA/IPA/BaseAddress  
    bits(64) end_address, // for range operations, end address  
    bits(2) tg,        // for range - the TG parameter  
    bits(4) ttl,       
  )

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_ALL

// TLBI_ALL()  
// =========  
// Invalidates all entries for the indicated translation regime with the  
// the indicated security state for all TLBs within the indicated shareability domain.  
// Invalidation applies to all applicable stage 1 and stage 2 entries.  
// The indicated attr defines the attributes of the memory operations that must be completed in  
// order to deem this operation to be completed.  
// When attr is TLBI ExcludeXS, only operations with XS=0 within the scope of this TLB operation  
// are required to complete.

TLBI ALL(SecurityState security, Regime regime, Shareability shareability, TLBI MemAttr attr)  
assert PSTATE.EL IN {EL3, EL2};  

TLBIRecord r;  
r.op          = TLBIOp_ALL;  
r.security    = security;  
r.regime      = regime;  
r.level       = TLBILevel_Any;  
r.attr        = attr;  

TLBI(r);  
if shareability != Shareability_None then Broadcast(shareability, r);  
return;
Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_ASID

// TLBI_ASID()
// ==========
// Invalidates all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
TLBI_MemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
r.op           = TLBIOp_ALL;
r.security     = security;
r.regime       = regime;
r.vmid         = vmid;
r.level        = TLBILevel_Any;
r.attr         = attr;
r.asid         = Xt<63:48>;
TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_IPAS2

// TLBI_IPAS2()
// ============
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2};

TLBIRecord r;
r.op           = TLBIOp_IPAS2;
r.security     = security;
r.regime       = regime;
r.vmid         = vmid;
r.level        = level;
r.attr         = attr;
r.ttl          = Xt<47:44>;

r.address.address = Xt<39:0> : Zeros(12);
r.address.NS      = if security == SS_NonSecure then '1' else Xt<63>;

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;
enumeration TLBI_MemAttr {
    TLBI_AllAttr,
    TLBI_ExcludeXS
};

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_MemAttr

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_RIPAS2

// TLBI_RIPAS2()
// =============
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
// When the indicated level is
//    TLBILevel_Any : this applies to TLB entries at all levels
//    TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)

    assert PSTATE.EL IN {EL3, EL2, EL1};
    TLBIRecord r;
    r.op           = TLBIOp_RIPAS2;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer baseaddr  = SInt(Xt<36:0>);
    bits(64) start_address;
    boolean valid;
    (valid, r.tg, start_address, r.end_address) = TLBI_Range(regime, Xt);
    if !valid then return;
    r.address.address = start_address<51:0>;
    TLBI(r);
    if shareability != Shareability_None then Broadcast(shareability, r);
    return;
Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_RVA

// TLBI_RVA()
// =========
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//    TLBILevel_Any : this applies to TLB entries at all levels
//    TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBIExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)  
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
  r.op = TLBIOp_RVA;
  r.security = security;
  r.regime = regime;
  r.vmid = vmid;
  r.level = level;
  r.attr = attr;
  r.asid = Xt<63:48>;
  r.ttl = Xt<47:44>;

bits(64) start_address;
boolean valid;

(valid, r.tg, start_address, r.end_address) = TLBI_Range(regime, Xt);

if !valid then return;

  r.address.address = start_address<51:0>;
  r.address.NS = if security == SS_NonSecure then '1' else Xt<63>;

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;
// TLBI_RVAA()
// ===========
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
// TLBI_LEVEL_Last : this applies to TLB entries at last level only
// TLBI_LEVEL_Any : this applies to TLB entries at all levels
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBIExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_RVAA(  
    SecurityState security,  
    Regime regime, bits(16) vmid,  
    Shareability shareability,  
    TLBILevel level,  
    TLBI_MemAttr attr,  
    bits(64) Xt)  
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op = TLBIOp_RVAA;
    r.security = security;
    r.regime = regime;
    r.vmid = vmid;
    r.level = level;
    r.attr = attr;
    r.ttl = Xt<47:44>;

    bits(2) tg = Xt<47:46>;
    integer scale = UInt(Xt<45:44>);
    integer num = UInt(Xt<43:39>);
    integer baseaddr = SInt(Xt<36:0>);

    bits(64) start_address;
    boolean valid;

    (valid, r.tg, start_address, r.end_address) = TLBI_Range(regime, Xt);

    if !valid then return;

    r.address.address = start_address<51:0>;
    TLBI(r);
    if shareability != Shareability_None then Broadcast(shareability, r);
    return;
// TLBI_Range()
// ============
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBI_Range(Regime regime, bits(64) Xt)
  boolean valid = TRUE;
  bits(64) start = Zeros(64);
  bits(64) end = Zeros(64);

  bits(2) tg = Xt<47:46>;
  integer scale = UInt(Xt<45:44>);
  integer num = UInt(Xt<43:39>);
  integer tg_bits;

  if tg == '00' then
    return (FALSE, tg, start, end);

  case tg of
    when '01' // 4KB
      tg_bits = 12;
      if HasLargeAddress(regime) then
        start<52:16> = Xt<36:0>;
        start<63:53> = Replicate(Xt<36>, 11);
      else
        start<48:12> = Xt<36:0>;
        start<63:49> = Replicate(Xt<36>, 15);
    when '10' // 16KB
      tg_bits = 14;
      if HasLargeAddress(regime) then
        start<52:16> = Xt<36:0>;
        start<63:53> = Replicate(Xt<36>, 11);
      else
        start<50:14> = Xt<36:0>;
        start<63:51> = Replicate(Xt<36>, 13);
    when '11' // 64KB
      tg_bits = 16;
      start<52:16> = Xt<36:0>;
      start<63:53> = Replicate(Xt<36>, 11);
    otherwise
      Unreachable();
  end

  integer range = (num+1) << (5*scale + 1 + tg_bits);
  end = start + range<63:0>;

  if end<52> != start<52> then
    // overflow, saturate it
    end = Replicate(start<52>, 64-52) : Ones(52);

  return (valid, tg, start, end);
TLBI_VA()  
// =========  
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain  
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime  
// with the indicated security state.  
// ASID, VA and related parameters are derived from Xt.  
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.  
// When the indicated level is  
//      TLBILevel_Any : this applies to TLB entries at all levels  
//      TLBILevel_Last : this applies to TLB entries at last level only  
// The indicated attr defines the attributes of the memory operations that must be completed in  
// order to deem this operation to be completed.  
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation  
// are required to complete.

assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
  r.op = TLBIOp_VA;
  r.security = security;
  r.regime = regime;
  r.vmid = vmid;
  r.level = level;
  r.attr = attr;
  r.asid = Xt<63:48>;
  r.ttl = Xt<47:44>;
  r.address.address = Xt<39:0> : Zeros(12);

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;

Shared Pseudocode Functions  
Page 3237
Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_VAA

// TLBI_VAA()
// ========
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//   TLBILevel_Any : this applies to TLB entries at all levels
//   TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBILevel level, TLBI_MemAttr attr, bits(64) Xt)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.ttl          = Xt<47:44>;
    r.address.address = Xt<39:0> : Zeros(12);
TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI_VMALL

// TLBI_VMALL()
// ============
// Invalidates all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.
TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBI_MemAttr attr)
assert PSTATE.EL IN {EL3, EL2, EL1};

TLBIRecord r;
    r.op           = TLBIOp_VMALL;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;
// TLBI_VMALLS12()
// ===============
// Invalidates all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
Shareability shareability, TLBI_MemAttr attr)
assert PSTATE.EL IN {EL3, EL2};
TLBIR e;
or.op = TLBIOp_VMALLS12;
or.security = security;
or.regime = regime;
or.level = TLBILevel_Any;
or.vmid = vmid;
or.attr = attr;

TLBI(r);
if shareability != Shareability_None then Broadcast(shareability, r);
return;

// VMID[]
// ======
// Effective VMID.

bits(16) VMID[]
if EL2Enabled() then
  return VTTBR_EL2.VMID;
elsif HaveEL(EL2) && HaveSecureEL2Ext() then
  return Zeros(16);
else
  return VMID_NONE;

// VMID_NONE
constant bits(16) VMID_NONE = Zeros();

// VBitOp
enumeration VBitOp {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};

// CompareOp

// ImmediateOp
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    boolean altfp = HaveAltFP() & !UsingAArch32() & FPCR.AH == '1';
    return Reduce(op, input, esize, altfp);

// Reduce()
// ========
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behaviour.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;
    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);

    case op of
      when ReduceOp_FMINNUM
          result = FPMinNum(lo, hi, FPCR[]);
      when ReduceOp_FMAXNUM
          result = FPMaxNum(lo, hi, FPCR[]);
      when ReduceOp_FMIN
          result = FPMin(lo, hi, FPCR[], altfp);
      when ReduceOp_FMAX
          result = FPMax(lo, hi, FPCR[], altfp);
      when ReduceOp_FADD
          result = FPAdd(lo, hi, FPCR[]);
      when ReduceOp_ADD
          result = lo + hi;

    return result;

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
    ReduceOp_FMIN, ReduceOp_FMAX,
    ReduceOp_FADD, ReduceOp_ADD};
AArch64.CombineS1S2Desc()

Combines the address descriptors from stage 1 and stage 2

AddressDescriptor AArch64.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc, AccType s2acctype)

AddressDescriptor result;
result.paddress = s2desc.paddress;
apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
if IsFault(s1desc) || IsFault(s2desc) then
    result = if IsFault(s1desc) then s1desc else s2desc;
else
    result.fault = AArch64.NoFault();
    if s2desc.memattrs.memtype == MemType_Device ||
       (apply_force_writeback && s1desc.memattrs.memtype == MemType_Device) then
        result.memattrs.device = s2desc.memattrs.device;
    elsif s2desc.memattrs.memtype == MemType_Normal then
        result.memattrs.device = s1desc.memattrs.device;
    else
        result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device, s2desc.memattrs.device);
    result.memattrs.tagged = FALSE;
    if s1 can be either Normal or Device, S2 is Normal.
else
    result.memattrs.memtype = MemType_Normal;
    result.memattrs.device = DeviceType UNKNOWN;
    result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner, s2acctype);
    result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer, s2acctype);
    result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
    result.memattrsoutershareable = (s1desc.memattrsoutershareable || s2desc.memattrsoutershareable);
    result.memattrs.tagged = (s1desc.memattrs.tagged &&
                           result.memattrs.inner.attrs == MemAttr_WB &&
                           result.memattrs.inner.hints == MemHint_RWA &&
                           result.memattrs.outer.attrs == MemAttr_WB &&
                           result.memattrs.outer.hints == MemHint_RWA);
result = MemAttrDefaults(result.memattrs);

return result;
Library pseudocode for aarch64/translation/attrs/AArch64.InstructionDevice

// AArch64.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                                  bits(52) ipaddress, integer level,
                                                  AccType acctype, boolean iswrite, boolean secondstage,
                                                  boolean s2fs1walk)

    c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch64.PermissionFault(ipaddress, boolean UNKNOWN, level, acctype, iswrite,
                                                  secondstage, s2fs1walk);
    else
        addrdesc.memattrs.memtype = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs.tagged = FALSE;
        addrdesc.memattrs = MemAttrDefaults(addrdesc.memattrs);

    return addrdesc;
MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;
    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    memattrs.tagged = FALSE;
    if ((attrfield<7:4> != '0000' && attrfield<7:4> != '1111' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);
    else
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

    if attrfield<7:4> == '0000' then            // Device
        memattrs.memtype = MemType_Device;
        case attrfield<3:0> of
            when '0000' memattrs.device = DeviceType_nGnRnE;
            when '0100' memattrs.device = DeviceType_nGnRE;
            when '1000' memattrs.device = DeviceType_nGRE;
            when '1100' memattrs.device = DeviceType_GRE;
            otherwise Unreachable();         // Reserved, handled above
        end;
    elsif attrfield<3:0> != '0000' then        // Normal
        memattrs.memtype = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    elsif HaveMTEExt() && attrfield == '11110000' then // Normal, Tagged WB-RWA
        memattrs.memtype = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints('1111', acctype); // WB_RWA
        memattrs.inner = LongConvertAttrsHints('1111', acctype); // WB_RWA
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
        memattrs.tagged = TRUE;
    else
        Unreachable();                          // Reserved, handled above

    if ((HCR_EL2.VM == '1' || HCR_EL2.DC == '1') &&
        (PSTATE.EL == EL1 || (PSTATE.EL == EL0 && HCR_EL2.TGE == '0')) &&
        acctype != AccType_NV2REGISTER) then
        return memattrs;
    else
        return MemAttrDefaults(memattrs);
// AArch64.TranslateAddressS1Off()
// ------------------------------
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
assert !ELUsingAArch32(S1TranslationRegime());

TLBRecord result;
result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;

Top = AddrTop(vaddress, (acctype == AccType_IFETCH), PSTATE.EL);
if !IsZero(vaddress<Top:PAMax()>)
level = 0;
ipaddress = bits(52) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;
result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,boolean UNKNOWN, level, acctype, iswrite, secondstage, s2fs1walk);
return result;

default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');
if default_cacheable then
// Use default cacheable settings
result.addrdesc.memattrs.memtype = MemType_Normal;
result.addrdesc.memattrs.inner.attrs = MemAttr_WB;  // Write-back
result.addrdesc.memattrs.inner.hints = MemHint_RWA;
result.addrdesc.memattrs.shareable = FALSE;
result.addrdesc.memattrs.outershareable = FALSE;
result.addrdesc.memattrs.tagged = HCR_EL2.DCT == '1';
elself acctype != AccType_IFETCH then
// Treat data as Device
result.addrdesc.memattrs.memtype = MemType_Device;
result.addrdesc.memattrs.device = DeviceType_nGnRnE;
result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
result.addrdesc.memattrs.tagged = FALSE;
elself
// Instruction cacheability controlled by SCTLR_ELx.I
cacheable = SCTLR[].I == '1';
result.addrdesc.memattrs.memtype = MemType_Normal;
if cacheable then
result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
result.addrdesc.memattrs.inner.hints = MemHint_RA;
elself
result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
result.addrdesc.memattrs.inner.hints = MemHint_No;
result.addrdesc.memattrs.shareable = TRUE;
result.addrdesc.memattrs.outershareable = TRUE;
result.addrdesc.memattrs.tagged = FALSE;
result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;
result.addrdesc.memattrs = MemAttrDefaults(result.addrdesc.memattrs);

result.perms.ap = bits(3) UNKNOWN;
result.perms.xn = '0';
result.perms.pxn = '0';

result.nG = bit UNKNOWN;
result.contiguous = boolean UNKNOWN;
result.domain = bits(4) UNKNOWN;
result.level = integer UNKNOWN;
result.blocksize = integer UNKNOWN;
result.addrdesc.paddress.address = vaddress<51:0>;
result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
result.addrdesc.fault = AArch64.NoFault();
result.descupdate.descaddr = result.addrdesc;
return result;

Library pseudocode for aarch64/translation/checks/AArch64.AccessUsesEL

// AArch64.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)
  if acctype == AccType_UNPRIV then
    return EL0;
  elsif acctype == AccType_NV2REGISTER then
    return EL2;
  elsif acctype == AccType_CSR_PRIV then
    assert PSTATE.EL == EL0;
    assert HaveCSRExt();
    if IsInHost() then
      return EL2;
    else
      return EL1;
  else
    return PSTATE.EL;
Library pseudocode for aarch64/translation/checks/AArch64.CheckPermission
FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level, bit NS, AccType acctype, boolean iswrite)

assert !ELUsingAArch32(S1TranslationRegime());

wxn = SCTLR[].WXN == '1';

if (PSTATE.EL == EL0 ||
    IsInHost() ||
    (PSTATE.EL == EL1 && !HaveNV2Ext()) ||
    (PSTATE.EL == EL1 && HaveNV2Ext() && (acctype != AccType_NV2REGISTER || !ELIsInHost(EL2)))) then
    priv_r = TRUE;
    priv_w = perms.ap<2> == '0';
    user_r = perms.ap<1> == '1';
    user_w = perms.ap<2:1> == '01';

    ispriv = AArch64.AccessUsesEL(acctype) != EL0;

    user_xn = perms.xn == '1' || (user_w && wxn);
    priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

if acctype IN {AccType_CSR_NORMAL, AccType_CSR_PRIV} then
    assert HaveCSRExt();

case PSTATE.EL of
    when EL0
        pan = CSRCR_EL0.PAN;
    when EL1
        pan = CSRCR_EL1.PAN;
    when EL2
        pan = CSRCR_EL2.PAN;
    otherwise
        Unreachable();
else
    pan = if HavePANExt() then PSTATE.PAN else '0';

if acctype IN {AccType_CSR_NORMAL, AccType_CSR_PRIV} then
    epan = '1';
else
    epan = if HavePAN3Ext() then SCTLR[].EPAN else '0';

// Restriction on Secure instruction fetch
if boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects PAN3 execute permission check" then
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        user_xn = TRUE;
    priv_xn = TRUE;

if (EL2Enabled()) && ((PSTATE.EL == EL1 && HaveNVExt() && HCR_EL2.<NV, NV1> == '11') ||
    (HaveNV2Ext() && acctype == AccType_NV2REGISTER && HCR_EL2.NV2 == '1')) then
    pan = '0';

    is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
    is_ats1xp = (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr());

if (pan == '1' && (user_r || (epan == '1' && !user_xn)) &&
    ispriv && (is_ldst || is_ats1xp)) then
    priv_r = FALSE;
    priv_w = FALSE;
else
    (r, w, xn) = (priv_r, priv_w, priv_xn);

else
    (r, w, xn) = (user_r, user_w, user_xn);

else // Access from EL2 or EL3
    r = TRUE;
    w = perms.ap<2> == '0';
    xn = perms.xn == '1' || (w && wxn);

// Restriction on Secure instruction fetch
if !boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects PAN3 execute permission check" then
if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
    xn = TRUE;

if acctype == AccType_IFETCH then
    fail = xn;
    failedread = TRUE;
elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW } then
    fail = !r || !w;
    failedread = !r;
elsif iswrite then
    fail = !w;
    failedread = FALSE;
elsif acctype == AccType_DC && PSTATE.EL != EL0 then
    // DC maintenance instructions operating by VA, cannot fault from stage 1 translation,
    // other than DC IVAC, which requires write permission, and operations executed at EL0,
    // which require read permission.
    fail = FALSE;
else
    fail = !r;
    failedread = TRUE;
if fail then
    secondstage = FALSE;
    s2fs1walk = FALSE;
    ipaddress = bits(52) UNKNOWN;
    return AArch64.PermissionFault(ipaddress,boolean UNKNOWN, level, acctype, !failedread, secondstage, s2fs1walk);
else
    return AArch64.NoFault();
FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(52) ipaddress, integer level, AccType acctype, boolean iswrite, boolean NS, boolean s2fs1walk, boolean hwupdatewalk)

assert (IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2)) && HasS2Translation();

r = perms.ap<1> == '1';
w = perms.ap<2> == '1';
if HaveExtendedExecuteNeverExt() then
  case perms.xn:perms.xxn of
    when '00'  xn = FALSE;
    when '01'  xn = PSTATE.EL == EL1;
    when '10'  xn = TRUE;
    when '11'  xn = PSTATE.EL == EL0;
  end;
else
  xn = perms.xn == '1';
// Stage 1 walk is checked as a read, regardless of the original type
if acctype == AccType_IFETCH && !s2fs1walk then
  fail = xn;
  failedread = TRUE;
elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW }) && !s2fs1walk then
  fail = !r || !w;
  failedread = !r;
elsif iswrite && !s2fs1walk then
  fail = !w;
  failedread = FALSE;
elsif acctype == AccType_DC && PSTATE.EL != EL0 && !s2fs1walk then
  // DC maintenance instructions operating by VA, with the exception of DC IVAC, do
  // not generate Permission instructions from stage 2 translation, other than when
  // performing a stage 1 translation table walk.
  fail = FALSE;
elsif hwupdatewalk then
  fail = !w;
  failedread = !iswrite;
else
  fail = !r;
  failedread = !iswrite;
if fail then
  domain = bits(4) UNKNOWN;
  secondstage = TRUE;
  return AArch64.PermissionFault(ipaddress,NS, level, acctype, !failedread, secondstage, s2fs1walk);
else
  return AArch64.NoFault();
Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;
    match = FALSE;
    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
        match = match || match_i;
    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();

Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)
    FaultRecord fault = AArch64.NoFault();
    d_side = (acctype != AccType_IFETCH);
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        mask = '0';
        generate_exception = AArch64.GenerateDebugExceptionsFrom(EL2, IsSecure(), mask) && MDSCR_EL1.MDE == '1';
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
        halt = HaltOnBreakpointOrWatchpoint();
    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, acctype, size);
        return fault;

Shared Pseudocode Functions
Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    if acctype IN {AccType_TTW, AccType_IC, AccType_AT} then
        return AArch64.NoFault();
    if acctype == AccType_DC then
        if !iswrite then
            return AArch64.NoFault();
        match = FALSE;
        ispriv = AArch64.AccessUsesEL(acctype) != EL0;
        for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
            match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite);
    if match && HaltOnBreakpointOrWatchpoint() then
        if acctype != AccType_NONFAULT && acctype != AccType_CNOTFIRST then
            reason = DebugHalt_Watchpoint;
            EDWAR = vaddress;
            Halt(reason);
        else
            // Fault will be reported and cancelled
            return AArch64.DebugFault(acctype, iswrite);
        elsif match then
            return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();

Library pseudocode for aarch64/translation/faults/AArch64.AccessFlagFault

// AArch64.AccessFlagFault()
// =========================

FaultRecord AArch64.AccessFlagFault(bits(52) ipaddress,boolean NS, integer level,
    AccType acctype, boolean iswrite, boolean secondstage,
    boolean s2fs1walk)
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress, NS, level, acctype, iswrite,
        extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.AddressSizeFault

// AArch64.AddressSizeFault()
// =========================

FaultRecord AArch64.AddressSizeFault(bits(52) ipaddress,boolean NS, integer level,
    AccType acctype, boolean iswrite, boolean secondstage,
    boolean s2fs1walk)
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress, NS, level, acctype, iswrite,
        extflag, errortype, secondstage, s2fs1walk);

Shared Pseudocode Functions
Library pseudocode for aarch64/translation/faults/AArch64.AlignmentFault

// AArch64.AlignmentFault()
// ========================

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.AsynchExternalAbort

// AArch64.AsynchExternalAbort()
// =============================

// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    faulttype = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(faulttype, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.DebugFault

// AArch64.DebugFault()
// ====================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(52) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);

Library pseudocode for aarch64/translation/faults/AArch64.ExclusiveFault

// AArch64.ExclusiveFault()
// ========================

FaultRecord AArch64.ExclusiveFault(AccType acctype, boolean iswrite)

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = boolean UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch64.CreateFaultRecord(Fault_Exclusive, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
Library pseudocode for aarch64/translation/faults/AArch64.NoFault

// AArch64.NoFault()
// ===============

FaultRecord AArch64.NoFault()

    ipaddress = bits(52) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress, boolean UNKNOWN, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
AArch64.CheckAndUpdateDescriptor

FaultRecord AArch64.CheckAndUpdateDescriptor(DescriptorUpdate result, FaultRecord fault, boolean secondstage, bits(64) vaddress, AccType acctype, boolean iswrite, boolean s2fs1walk, boolean hwupdatewalk)

boolean hw_update_AF = FALSE;
boolean hw_update_AP = FALSE;

// Check if access flag can be updated
// Address translation instructions are permitted to update AF but not required
if result.AF then
    if fault.statusCode == Fault_None || ConstraineUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE then
        hw_update_AF = TRUE;
    end if
endif

if result.AP && fault.statusCode == Fault_None then
    write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW}) && !s2fs1walk;
    hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT, AccType_DC, AccType_DC_UNPRIV})) || hwupdatewalk;
endif

if hw_update_AF || hw_update_AP then
    if secondstage || !HasS2Translation() then
        descaddr2 = result.descaddr;
    else
        hwupdatewalk = TRUE;
        descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
        if IsFault(descaddr2) then
            return descaddr2.fault;
        end if
    end if
endif

accdesc = CreateAccessDescriptor(AccType_ATOMICRW, FALSE);
desc = Mem[descaddr2, 8, accdesc, iswrite];
case el of
    when EL3
        reversedescriptors = SCTLR_EL3.EE == '1';
    when EL2
        reversedescriptors = SCTLR_EL2.EE == '1';
    otherwise
        reversedescriptors = SCTLR_EL1.EE == '1';
    end case
if reversedescriptors then
    desc = BigEndianReverse(desc);
endif

if hw_update_AF then
    desc<10> = '1';
else
    desc<7> = (if secondstage then '1' else '0');
end if

_Mem[descaddr2, 8, accdesc] = if reversedescriptors then BigEndianReverse(desc) else desc;
return fault;
Library pseudocode for aarch64/translation/translation/AArch64.FirstStageTranslate

// AArch64.FirstStageTranslate()
// =============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
  s1_enabled = SCTLR_EL2.M == '1';
else if HasS2Translation() then
  s1_enabled = HCR_EL2.TGE == '0' & HCR_EL2.DC == '0' & SCTLR_EL1.M == '1';
else
  s1_enabled = SCTLR[].M == '1';

TLBRecord S1;
S1.addrdesc.fault = AArch64.NoFault();
ipaddress = bits(52) UNKNOWN;
secondstage = FALSE;
s2fs1walk = FALSE;

if s1_enabled then // First stage enabled
  S1 = AArch64.TranslationTableWalk(ipaddress, TRUE, vaddress, acctype, iswrite, secondstage, s2fs1walk, size);
  permissioncheck = TRUE;
  if acctype == AccType_IFETCH then
    InGuardedPage = S1.GP == '1'; // Global state updated on instruction fetch that denotes
                                // if the fetched instruction is from a guarded page.
  else
    S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
    permissioncheck = FALSE;
    InGuardedPage = FALSE; // No memory is guarded when stage 1 address translation is disabled
  endif
endif

if !IsFault(S1.addrdesc) && UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() && nTLSMD = if S1TranslationRegime() == EL2 then SCTLR_EL2.nTLSMD else SCTLR_EL1.nTLSMD; then
  S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
endif

if (!wasaligned && acctype != AccType_IFETCH) || acctype == AccType_DCZVA && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device then
  S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
endif

if !IsFault(S1.addrdesc) && permissioncheck then
  S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level, S1.addrdesc.paddress.NS, acctype, iswrite);
endif

// Check for instruction fetches from Device memory not marked as execute-never. If there has
// not been a Permission Fault then the memory is not marked execute-never.
if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device && acctype == AccType_IFETCH) then
  S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level, acctype, iswrite, secondstage, s2fs1walk);
endif

// Check and update translation table descriptor if required
hwupdatewalk = FALSE;
s2fs1walk = FALSE;
S1.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S1.descupdate, S1.addrdesc.fault, secondstage, vaddress, acctype, iswrite, s2fs1walk, hwupdatewalk);

return S1.addrdesc;
// AArch64.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)

    // First Stage Translation
    S1 = AArch64.FirstStageTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(S1) && !HaveNV2Ext() && acctype == AccType_NV2REGISTER) && HasS2Translation() then
        s2fs1walk = FALSE;
        hwupdatewalk = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                               size, hwupdatewalk);
    else
        result = S1;
    return result;
// AArch64.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, boolean s2fs1walk, integer size, boolean hwupdatewalk)

assert HasS2Translation();

s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
secondstage = TRUE;

if s2_enabled then // Second stage enabled
  ipaddress = S1.paddress.address<51:0>;
  NS = S1.paddress.NS == '1';
  S2 = AArch64.TranslationTableWalk(ipaddress, NS, vaddress, acctype, iswrite, secondstage, s2fs1walk, size);

  // Check for unaligned data accesses to Device memory
  if (((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA && !s2fs1walk))
    && S2.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S2.addrdesc)) then
    S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

  // Check for permissions on Stage2 translations
  if !IsFault(S2.addrdesc) then
    S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level, acctype, iswrite, NS, s2fs1walk, hwupdatewalk);

  // Check for instruction fetches from Device memory not marked as execute-never. As there
  // has not been a Permission Fault then the memory is not marked execute-never.
  if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.memtype == MemType_Device && acctype == AccType_IFETCH) then
    S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level, acctype, iswrite, secondstage, s2fs1walk);

  if (s2fs1walk && !IsFault(S2.addrdesc) &&
    S2.addrdesc.memattrs.memtype == MemType_Device) then
    // Check for protected table walk.
    if HCR_EL2.PTW == '1' then
      S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, NS, S2.level, acctype, iswrite, secondstage, s2fs1walk);

    else // Translation table walk occurs as Normal Non-cacheable memory.
      S2.addrdesc.memattrs.memtype = MemType_Normal;
      S2.addrdesc.memattrs.inner.attrs = MemAttr_NC;
      S2.addrdesc.memattrs.outer.attrs = MemAttr_NC;
      S2.addrdesc.memattrs.shareable = TRUE;
      S2.addrdesc.memattrs.outershareable = TRUE;

    // Check and update translation table descriptor if required
    S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault, secondstage, vaddress, acctype, iswrite, s2fs1walk, hwupdatewalk);

    if s2fs1walk then
      result = AArch64.CombineS1S2Desc(S1, S2.addrdesc, AccType_TTW);
    else
      result = AArch64.CombineS1S2Desc(S1, S2.addrdesc, acctype);
  else
    result = S1;

return result;
Library pseudocode for aarch64/translation/translation/AArch64.SecondStageWalk

// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype, boolean iswrite, integer size, boolean hwupdatewalk)

    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk, size, hwupdatewalk);

Library pseudocode for aarch64/translation/translation/AArch64.TranslateAddress

// AArch64.TranslateAddress()
// =========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite, boolean wasaligned, integer size)

    result = AArch64.FullTranslate(vaddress, acctype, iswrite, wasaligned, size);
    if !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
AArch64.TranslationTableWalk()

Returns a result of a translation table walk

Implementations might cache information from memory in any number of non-coherent TLB caching structures, and so avoid memory accesses that have been expressed in this pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(52) ipaddress, boolean s1_nonsecure, bits(64) vaddress, AccType accetype, boolean iswrite, boolean secondstage, boolean s2fs1walk, integer size)

if !secondstage then
    assert !ELUsingAArch32(S1TranslationRegime());
else
    assert (IsSecureEL2Enabled() || (HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2))) && HasS2Translation();

TLBRecord result;
AddressDescriptor descaddr;
bits(64) baseregister;
bits(64) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2
bit nswalk;                    // Stage 2 translation table walks are to Secure or to Non-secure PA space

result.descupdate.AF = FALSE;
result.descupdate.AP = FALSE;
descaddr.memattrs.memtype = MemType_Normal;

// Derived parameters for the translation table walk:
//  grainsize = Log2(Size of Table) - Size of Table is 4KB, 16KB or 64KB in AArch64
//  stride = Log2(Address per Level) - Bits of address consumed at each level
//  firstblocklevel = First level where a block entry is allowed
//  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
//  inputsize = Log2(Size of Input Address) - Input Address size in bits
//  level = Level to start walk from
// This means that the number of levels after start level = 3-level

if !secondstage then
    // First stage translation
    inputaddr = ZeroExtend(vaddress);
    el = AArch64.AccessUsesEL(accetype);
    isprivileged = AArch64.AccessUsesUsesEL(accetype) != EL0;
top = AddrTop(inputaddr, (accetype == AccType_IFETCH), el);
    if el == EL3 then
        largegrain = TCR_EL3.TG0 == '01';
        midgrain = TCR_EL3.TG0 == '10';
        inputsize = 64 - UInt(TCR_EL3.T0SZ);
        inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
        if !Have52BitVAExt() && inputsize > inputsize_max then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_max;
        if inputsize < inputsize_min then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_min;
        ps = TCR_EL3.PS;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top>);
        disabled = FALSE;
        baseregister = TTBR0_EL3;
        descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
        reversedescriptors = SCTLR_EL3.EE == '1';
        lookupsecure = TRUE;
        singlepriv = TRUE;
        update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
        hierattrdisabled = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
    elsif ELIsInHost(el) then
        if inputaddr<top> == '0' then
            Shared Pseudocode Functions
            Page 3261
reversedescriptors = SCTLR_EL2.EE == '1';
lookupsecure = if IsSecureEL2Enabled() then IsSecure() else FALSE;
singlepriv = TRUE;
update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
hierattrsdissabled = AArch64.HaveHPDEXT() && TCR_EL2.HPD == '1';

else if inputaddr<top> == '0' then
    inputsize = 64 - UInt(TCR_EL1.T0SZ);
largegrain = TCR_EL1.TG0 == '01';
midgrain = TCR_EL1.TG0 == '10';
    inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;

    if !Have52BitVAExt() && inputsize > inputsize_max then
        c = ConstranUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_max;

    inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 52 else 48);
    if inputsize < inputsize_min then
        c = ConstranUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_min;
        basefound = inputsize <= inputsize_max && inputsize <= inputsize_max && IsZero(inputaddr<top>)

    if el == EL0 && TCR_EL1.NFD0 == '1' then
        disabled = disabled || acctype == AccType_NONFAULT || (IsDataAccess(acctype) && HaveTME());
        baseregister = TTRB0_EL1;
descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0, secondstage);
    hierattrsdissabled = AArch64.HaveHPDEXT() && TCR_EL1.HPD0 == '1';
else
    inputsize = 64 - UInt(TCR_EL1.T1SZ);
largegrain = TCR_EL1.TG1 == '11'; // TG1 and TG0 encodings differ
    inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;

    if !Have52BitVAExt() && inputsize > inputsize_max then
        c = ConstranUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_max;

    inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 47 else 48);
    if inputsize < inputsize_min then
        c = ConstranUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_min;
        basefound = inputsize <= inputsize_max && inputsize <= inputsize_max && IsZero(inputaddr<top>);
        if !Have52BitVAExt() && inputsize > inputsize_max then
            c = ConstranUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_max;

    inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 47 else 48);
    if inputsize < inputsize_min then
        c = ConstranUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_FAULT};
        if c == Constraint_FORCE then inputsize = inputsize_min;
        basefound = inputsize <= inputsize_max && inputsize <= inputsize_max && IsZero(inputaddr<top>);

    if el == EL0 && TCR_EL1.NFD1 == '1' then
        disabled = disabled || acctype == AccType_NONFAULT || (IsDataAccess(acctype) && HaveTME());
        baseregister = TTRB1_EL1;
descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1, secondstage);
    hierattrsdissabled = AArch64.HaveHPDEXT() && TCR_EL1.HPD1 == '1';

ps = TCR_EL1.IPS;
reversedescriptors = SCTLR_EL1.EE == '1';
lookupsecure = IsSecure();
singlepriv = FALSE;
update_AF = HaveAccessFlagUpdateExt() && TCR_EL1.HA == '1';
update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL1.HD == '1';

if largegrain then
    grainsize = 16; // Log2(64KB page size)
    firstblocklevel = (if Have52BitPAExt() then 1 else 2); // Largest block is 4TB (2^42 bytes)
    stride = grainsize - 3; // and 512MB (2^29 bytes) otherwise
elsif midgrain then
    grainsize = 14; // Log2(16KB page size)
    firstblocklevel = 2; // Largest block is 32MB (2^25 bytes)
else // Small grain
    grainsize = 12; // Log2(4KB page size)
    firstblocklevel = 1; // Largest block is 1GB (2^30 bytes)
    stride = grainsize - 3; // Log2(page size / 8 bytes)
The starting level is the number of strides needed to consume the input address level = 4 - (1 + (inputsize - grainsize - 1) DIV stride));

else
  // Second stage translation
  inputaddr = ZeroExtend(ipaddress);
  if IsSecureBelowEL3() then
    // Second stage for Secure translation regime
    if s1_nonsecure then
      // Non-secure IPA space
      t0size = VTCR_EL2.T0SZ;
      tg0 = VTCR_EL2.TG0;
      nswalk = VTCR_EL2.NSW;
    else
      // Secure IPA space
      t0size = VSTCR_EL2.T0SZ;
      tg0 = VSTCR_EL2.TG0;
      nswalk = VSTCR_EL2.SW;

    // Stage 2 translation accesses the Non-secure PA space or the Secure PA space
    if nswalk == '1' then
      // When walk is Non-secure, access must be to the Non-secure PA space
      nsaccess = '1';
    elsif !s1_nonsecure then
      // When walk is Secure and in the Secure IPA space,
      // access is specified by VSTCR_EL2.SA
      nsaccess = VSTCR_EL2.SA;
    elsif VSTCR_EL2.SW == '1' || VSTCR_EL2.SA == '1' then
      // When walk is Secure and in the Non-secure IPA space,
      // access is Non-secure when VSTCR_EL2.SA specifies the Non-secure PA space
      nsaccess = '1';
    else
      // When walk is Secure and in the Non-secure IPA space,
      // if VSTCR_EL2.SA specifies the Secure PA space, access is specified by VTCR_EL2.NSA
      nsaccess = VTCR_EL2.NSA;
  else
    // Second stage for Non-secure translation regime
    t0size = VTCR_EL2.T0SZ;
    tg0 = VTCR_EL2.TG0;
    nswalk = '1';
    nsaccess = '1';

  inputsize = 64 - UInt(t0size);
  largegrain = tg0 == '01';
  midgrain = tg0 == '10';

  inputsize_max = if Have52BitPAExt() && PAMax() == 52 && largegrain then 52 else 48;
  if !Have52BitPAExt() && inputsize > inputsize_max then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_max;

  inputsize_min = 64 - (if !HaveSmallTranslationTableExt() then 39 else if largegrain then 47 else 48);
  if inputsize < inputsize_min then
    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_min;

  ps = VTCR_EL2.PS;
  basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<63:inputsize>);
  descaddr.memattrs = WalkAttrDecode(VTCR_EL2.SH0, VTCR_EL2.ORGN0, VTCR_EL2.IRGN0, secondstage);
  reversedescriptors = SCTLR_EL2.EE == '1';
  singlepriv = TRUE;
  update_AF = HaveAccessFlagUpdateExt() && VTCR_EL2.HA == '1';
  update_AP = HaveDirtyBitModifierExt() && update_AF && VTCR_EL2.HD == '1';

  if IsSecureEL2Enabled() then
    lookupsecure = !s1_nonsecure;
  else
    lookupsecure = FALSE;

  if lookupsecure then
baseregister = VSTTBR_EL2;
startlevel = UInt(VSTCR_EL2.SL0);

else
baseregister = VTTBR_EL2;
startlevel = UInt(VTCR_EL2.SL0);

if largegrain then
  grainsize = 16; // Log2(64KB page size)
  level = 3 - startlevel;
  firstblocklevel = (if Have52BitPAExt() then 1 else 2); // Largest block is 4TB (2^42 bytes)
  // and 512MB (2^29 bytes) otherwise
else if midgrain then
  grainsize = 14; // Log2(16KB page size)
  level = 3 - startlevel;
  firstblocklevel = 2; // Largest block is 32MB (2^25 bytes)
else
  grainsize = 12; // Log2(4KB page size)
  if HaveSmallTranslationTableExt() && startlevel == 3 then
    level = startlevel; // Startlevel 3 (VTCR_EL2.SL0 or VSTCR_EL2.SL0 == 0b11) for 4KB granule
  else
    level = 2 - startlevel;
  firstblocklevel = 1; // Largest block is 1GB (2^30 bytes)
  stride = grainsize - 3; // Log2(page size / 8 bytes)

  // Limits on IPA controls based on implemented PA size. Level 0 is only
  // supported by small grain translations
  if largegrain then // 64KB pages
    // Level 1 only supported if implemented PA size is greater than 2^42 bytes
    if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
  elsif midgrain then // 16KB pages
    // Level 1 only supported if implemented PA size is greater than 2^40 bytes
    if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
  else // Small grain, 4KB pages
    // Level 0 only supported if implemented PA size is greater than 2^42 bytes
    if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;
  
  // If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
  inputsizecheck = inputsize;
  if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
    case ConstrainUnpredictable(Unpredictable_LARGEIPA) of
      when Constraint_FORCE // Restrict the inputsize to the PAMax value
        inputsize = PAMax();
        inputsizecheck = PAMax();
      when Constraint_FORCENOSLCHECK // As FORCE, except use the configured inputsize in the size checks below
        inputsize = PAMax();
      when Constraint_FAULT // Generate a translation fault
        basefound = FALSE;
      otherwise // Unreachable()
        inputsizecheck = inputsize;
  
  // Number of entries in the starting level table =
  // (Size of Input Address)/(Address per level)^(Num levels remaining)*(Size of Table))
  startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

  // Check for starting level table with fewer than 2 entries or longer than 16 pages.
  // Lower bound check is: startsizecheck < Log2(2 entries)
  // Upper bound check is: startsizecheck > Log2(pagesize/8*16)
  if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

  if !basefound || disabled then
    level = 0; // AArch32 reports this as a level 1 fault
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite,
        secondstage, s2fs1walk);
  
  return result;

  case ps of
    when '000' outputsize = 32;
    when '001' outputsize = 36;
    when '010' outputsize = 40;
when '011' outputsize = 42;
when '100' outputsize = 44;
when '101' outputsize = 48;
when '110' outputsize = (if Have52BitPAExt() && largegrain then 52 else 48);
otherwise outputsize = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

if outputsize > PAMax() then outputsize = PAMax();

if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
  level = 0;
  result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
  return result;

// Bottom bound of the Base address is:
// Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
// Number of entries in starting level table =
// (Size of Input Address)/(Address per level)^((Num levels remaining)*(Size of Table))
baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
if outputsize == 52 then
  z = (if baselowerbound < 6 then 6 else baselowerbound);
basedaddress = baseregister<5:2>:baseregister<47:z>:Zeros(z);
else
  baseaddress = ZeroExtend(baseregister<47:baselowerbound>:Zeros(baselowerbound));

ns_table = if lookupsecure then '0' else '1';
ap_table = '00';
xn_table = '0';
pxn_table = '0';
addrselecttop = inputsize - 1;

apply_nvnv1_effect = HaveNVExt() && EL2Enabled() && HCR_EL2.<NV,NV1> == '11' &&
     S1TranslationRegime() ||
   repeat
     addrselectbottom = (3-level)*stride + grainsize;
     bits(52) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:='000');
     descaddr.paddress.address = baseaddress OR index;
     descaddr.paddress.NS = if secondstage then nswalk else ns_table;

     // If there are two stages of translation, then the first stage table walk addresses
     // are themselves subject to translation
     if secondstage || !HasS2Translation() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) then
        descaddr2 = descaddr;
     else
        hwupdatewalk = FALSE;
        descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
        // Check for a fault on the stage 2 walk
        if IsFault(descaddr2) then
            result.addrdesc.fault = descaddr2.fault;
            return result;

     // Update virtual address for abort functions
     descaddr2.vaddress = ZeroExtend(vaddress);
     accdesc = CreateAccessDescriptorTTW(acctype, secondstage, s2fs1walk, level);
     desc = _Mem[descaddr2, 8, accdesc, iswrite];

     if reversedescriptors then desc = BigEndianReverse(desc);
     if desc<0> == '0' || (desc<1:0> == '01' && (level == 3 ||
      // Fault (00), Reserved (10), Block (01) at level 3, or Block(01) with nT bit set.
      result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
      return result;

     // Valid Block, Page, or Table entry
     if desc<1:0> == '01' || level == 3 then
        blocktranslate = TRUE;
else
  // Table (11)
  if (outputsize < 52 && largegrain && (PAMax() == 52 || boolean IMPLEMENTATION_DEFINED "Address Size Fault on LPA descriptor bits [15:12]") && !IsZero(desc<15:12>)) || (outputsize < 48 && !IsZero(desc<47:outputsize>)) then
    result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);

  return result;

if outputsize == 52 then
  baseaddress = desc<15:12>:desc<47:grainsize>:Zeros(grainsize);
else
  baseaddress = ZeroExtend(desc<47:grainsize>:Zeros(grainsize));

if !secondstage then
  // Unpack the upper and lower table attributes
  ns_table    = ns_table    OR desc<63>;
if !secondstage && !hierattrsdisabled then
  ap_table<1> = ap_table<1> OR desc<62>;
    // read-only
if apply_nvnv1_effect then
  pxn_table    = pxn_table    OR desc<60>;
else
  xn_table    = xn_table    OR desc<60>;
// pxn_table and ap_table[0] apply in EL1&0 or EL2&0 translation regimes
if !singlepriv then
    if !apply_nvnv1_effect then
      pxn_table    = pxn_table    OR desc<59>;
      ap_table<0> = ap_table<0> OR desc<61>;
        // privileged

  level = level + 1;
  addrselecttop = addrselectbottom - 1;
  blocktranslate = FALSE;
until blocktranslate;

// Check block size is supported at this level
if level < firstblocklevel then
  result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);

  return result;

// Check for misprogramming of the contiguous bit
if largegrain then
  num_ch_entries = 5;
elsif midgrain then
  num_ch_entries = if level == 3 then 7 else 5;
else
  num_ch_entries = 4;

contiguousbitcheck = inputsize < (addrselectbottom + num_ch_entries);

if contiguousbitcheck && desc<52> == '1' then
  if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
    result.addrdesc.fault = AArch64.TranslationFault(ipaddress, s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);

    return result;

// Unpack the descriptor into address and upper and lower block attributes
if largegrain then
  outputaddress = desc<15:12>:desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;
else
  outputaddress = ZeroExtend(desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>);

// When 52-bit PA is supported, for 64 Kbyte translation granule,
// block size might be larger than the supported output address size
if ((outputsize < 52 && !IsZero(outputaddress<51:48>) && largegrain && (PAMax() == 52 || boolean IMPLEMENTATION_DEFINED "Address Size Fault on LPA descriptor bits [15:12]"))) || (outputsize < 48 && !IsZero(outputaddress<47:outputsize>)) then
  result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,s1_nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);

  return result;
// Check Access Flag
if desc<10> == '0' then
  if !update_AF then
    result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress, sl nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;
  else
    result.descupdate.AF = TRUE;
  end if
elseif update_AP && desc<51> == '1' then
  if !secondstage && desc<7> == '1' then
    desc<7> = '0';
  else if secondstage && desc<7> == '0' then
    desc<7> = '1';
  end if
  result.descupdate.AP = TRUE;
end if
// Required descriptor if AF or AP[2]/S2AP[2] needs update
result.descupdate.descaddr = descaddr;

if apply_nvnl_effect then
  pnxn = desc<54>;
  xn = '0';
  ap = desc<7>:'01';
else
  xn = desc<54>;
  pnxn = desc<53>;
  ap = desc<7:6>:'1';
end if
contiguousbit = desc<52>;
nG = desc<11>;
sh = desc<9:8>;
memattr = desc<5:2>;
// AttrIndx and NS bit in stage 1
result.domain = bits(4) UNKNOWN;
result.level = level;
result.blocksize = 2^((3-level)*stride + grainsize);
// Stage 1 translation regimes also inherit attributes from the tables
if !secondstage then
  result.perms.xn      = xn OR xn_table;
  result.perms.ap<2>   = ap<2> OR ap_table<1>; // Force read-only
  if !singlepriv then
    result.perms.ap<1> = ap<1> AND NOT(ap_table<0>); // Force privileged only
    result.perms.pxn   = pnx OR pnxn_table;
      // Pages from Non-secure tables are marked non-global in Secure EL1&0
    if !IsSecure() then
      result.nG = nG OR ns_table;
    else
      result.nG = nG;
    end if
  else
    result.perms.ap<1> = '1';
    result.perms.pxn   = '0';
    result.nG           = '0';
    result.GP          = desc<50>;
    result.perms.ap<0>  = '1';
    result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
    result.addrdesc.paddress.NS = memattr<3> OR ns_table;
  end if
else
  result.perms.ap<2:1> = ap<2:1>;
  result.perms.ap<0>  = '1';
  result.perms.xn     = xn;
  if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
  end if
  if s2fs1walk then
    result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_TTW);
  else
    result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
  end if
result.addrdesc.paddress.NS = nsaccess;
result.addrdesc.paddress.address = outputaddress;
result.addrdesc.fault = AArch64_NoFault();
result.contiguous = contiguousbit == '1';
if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;
return result;

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

// ClearStickyErrors()
// ===============
ClearStickyErrors()
EDSCR.TXU = '0';            // Clear TX underrun flag
EDSCR.RXO = '0';            // Clear RX overrun flag
if Halted() then            // in Debug state
    EDSR.ITO = '0';        // Clear ITR overrun flag

// If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSR.ERR is cleared.
// The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
// in the pseudocode.
if Halted() && EDSR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then return;
EDSCR.ERR = '0';            // Clear cumulative error flag
return;

Library pseudocode for shared/debug/DebugTarget/DebugTarget

// DebugTarget()
// ===========
// Returns the debug exception target Exception level
bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

// DebugTargetFrom()
// ===============
bits(2) DebugTargetFrom(boolean secure)
if HaveEL(EL2) && (!secure || (HaveSecureEL2Ext() &&
    (HaveEL(EL3) || SCR_EL3.EEL2 == '1'))) then
    if ELUsingAArch32(EL2) then
        route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
    else
        route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;
if route_to_el2 then
    target = EL2;
elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
    target = EL3;
else
    target = EL1;
return target;
// DoubleLockStatus()
// =====================
// Returns the state of the OS Double Lock.
//   FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//   TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
if !HaveDoubleLock() then
    return FALSE;
elsif ELUsingAArch32(EL1) then
    return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
else
    return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();

// AllowExternalDebugAccess()
// =========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
if HaveSecureExtDebugView() then
    return AllowExternalDebugAccess(IsAccessSecure());
else
    return AllowExternalDebugAccess(ExternalSecureInvasiveDebugEnabled());

// AllowExternalDebugAccess()
// =========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(boolean allow_secure)
if HaveSecureExtDebugView() || ExternalInvasiveDebugEnabled() then
    if allow_secure then
        return TRUE;
    elsif HaveEL(EL3) then
        if ELUsingAArch32(EL3) then
            return SDCR.EDAD == '0';
        else
            return MDCR_EL3.EDAD == '0';
        else
            return !IsSecure();
    else
        return FALSE;
Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

// AllowExternalPMUAccess()
// ================
// Returns TRUE if an external debug interface access to the PMU registers is allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
// The access may also be subject to OS Lock, power-down, etc.
  if HaveSecureExtDebugView() then
    return AllowExternalPMUAccess(IsAccessSecure());
  else
    return AllowExternalPMUAccess(ExternalSecureNoninvasiveDebugEnabled());

// AllowExternalPMUAccess()
// ================
// Returns TRUE if an external debug interface access to the PMU registers is allowed for the given
// Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(boolean allow_secure)
// The access may also be subject to OS Lock, power-down, etc.
  if HaveSecureExtDebugView() || ExternalNoninvasiveDebugEnabled() then
    if allow_secure then
      return TRUE;
    elsif HaveEL(EL3) then
      if ELUsingAArch32(EL3) then
        return SDCR.EPMAD == '0';
      else
        return MDCR_EL3.EPMAD == '0';
      end
    else
      return !IsSecure();
    end
  else
    return FALSE;

Library pseudocode for shared/debug/authentication/AllowExternalTraceAccess

// AllowExternalTraceAccess()
// ================
// Returns TRUE if an external Trace access to the Trace registers is allowed, FALSE otherwise.

boolean AllowExternalTraceAccess()
// The access may also be subject to OS Lock, power-down, etc.
  if !HaveTraceBufferExtension() then
    return TRUE;
  else
    return AllowExternalTraceAccess(IsAccessSecure());

// AllowExternalTraceAccess()
// ================
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given Security state, FALSE otherwise.

boolean AllowExternalTraceAccess(boolean access_is_secure)
// The access may also be subject to OS Lock, power-down, etc.
  if !HaveTraceBufferExtension() || access_is_secure then
    return TRUE;
  elsif HaveEL(EL3) then
    // External Trace access is not supported for EL3 using AArch32
    assert !ELUsingAArch32(EL3);
    return MDCR_EL3.ETAD == '0';
  else
    return !IsSecure();

Library pseudocode for shared/debug/authentication/Debug_authentication

signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

// ExternalInvasiveDebugEnabled()
// ****************************
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == HIGH;

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

// ExternalNoninvasiveDebugAllowed()
// ---------------------------------
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||
             (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1'));

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

// ExternalNoninvasiveDebugEnabled()
// ---------------------------------
// This function returns TRUE if the FEAT_Debugv8p4 is implemented, otherwise this
// function is IMPLEMENTATION DEFINED.
// In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
// OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return !HaveNoninvasiveDebugAuth() ||
           ExternalInvasiveDebugEnabled() || NIDEN == HIGH;

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

// ExternalSecureInvasiveDebugEnabled()
// -----------------------------------
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

// ExternalSecureNoninvasiveDebugEnabled()
// --------------------------------------
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    if !HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureInvasiveDebugEnabled();

Library pseudocode for shared/debug/authentication/IsAccessSecure

// Returns TRUE when an access is Secure
boolean IsAccessSecure();
Library pseudocode for shared/debug/authentication/IsCorePowered

// Returns TRUE if the Core power domain is powered on, FALSE otherwise.  
boolean IsCorePowered();

Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

// CheckValidStateMatch()  
--------------------------  
// Checks for an invalid state match that will generate Constrained Unpredictable behaviour, otherwise  
// returns Constraint_NONE.  

(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean isbreakpnt)  
boolean reserved = FALSE;  
// Match 'Usr/Sys/Svc' only valid for AArch32 breakpoints  
if (!isbreakpnt || !HaveAArch32EL(EL1)) & HMC:PxC == '000' & SSC != '11' then  
reserved = TRUE;  
// Both EL3 and EL2 are not implemented  
if !HaveEL(EL3) & !HaveEL(EL2) & (HMC != '0' || SSC != '00') then  
reserved = TRUE;  
// EL3 is not implemented  
if !HaveEL(EL3) & SSC IN {'01','10'} & HMC:SSC:PxC != '10100' then  
reserved = TRUE;  
// EL2 is not implemented  
if !HaveEL(EL2) & HMC:SSC:PxC == '11100' then  
reserved = TRUE;  
// Secure EL2 is not implemented  
if !HaveSecureEL2Ext() & (HMC:SSC:PxC) IN {'01110','100x0','10110','11x10'} then  
reserved = TRUE;  
if reserved then  
// If parameters are set to a reserved type, behaves as either disabled or a defined type  
(c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);  
assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};  
if c == Constraint_DISABLED then  
return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);  
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value  
return (Constraint_NONE, SSC, HMC, PxC);

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

// Set a Cross Trigger multi-cycle input event trigger to the specified level.  
CTI_SetEventLevel(CrossTriggerIn id, signal level);

Library pseudocode for shared/debug/cti/CTI_SignalEvent

// Signal a discrete event on a Cross Trigger input event trigger.  
CTI_SignalEvent(CrossTriggerIn id);
Library pseudocode for shared/debug/cti/CrossTrigger

```
enum CrossTriggerOut { CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest, CrossTriggerOut_IRQ, CrossTriggerOut_RSVD3, CrossTriggerOut_TraceExtIn0, CrossTriggerOut_TraceExtIn1, CrossTriggerOut_TraceExtIn2, CrossTriggerOut_TraceExtIn3);
```

```
enum CrossTriggerIn  { CrossTriggerIn_CrossHalt, CrossTriggerIn_PMUOverflow, CrossTriggerIn_RSVD2, CrossTriggerIn_RSVD3, CrossTriggerIn_TraceExtOut0, CrossTriggerIn_TraceExtOut1, CrossTriggerIn_TraceExtOut2, CrossTriggerIn_TraceExtOut3};
```

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================
CheckForDCCInterrupts()
commrx = (EDSCR.RXfull == '1');
commtx = (EDSCR.TXfull == '0');

// COMMRX and COMMTX support is optional and not recommended for new designs.
// SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
// SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

// The value to be driven onto the common COMMIRQ signal.
if ELUsingAArch32(EL1) then
  commirq = ((commrx && DBGDCCINT.RX == '1') || (commtx && DBGDCCINT.TX == '1'));
else
  commirq = ((commrx && MDCCINT_EL1.RX == '1') || (commtx && MDCCINT_EL1.TX == '1'));
SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);
return;
```
// DBGDTRRX_EL0[] (external write)  
// =========================================================================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

if EDPRSR<6:5,0> != '001' then
  IMPLEMENTATION_DEFINED "generate error response";
  return;
endif

if EDSCR.ERR == '1' then return; // Error flag set: ignore write

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
endif

if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
  EDSCR.RXO = '1';  EDSCR.ERR = '1'; // Overrun condition: ignore write
  return;
endif

EDSCR.RXfull = '1';
DTRRX = value;

if Halted() && EDSCR.MA == '1' then
  EDSCR.ITE = '0'; // See comments in EDITR[] (external write)
  if UsingAArch32() then
    ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_EL0"
    ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
    X[1] = bits(64) UNKNOWN;
  else
    ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
    ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
    R[1] = bits(32) UNKNOWN;
  endif
  EDSCR.RXfull = bit UNKNOWN;
  DBGDTRRX_EL0 = bits(64) UNKNOWN;
else
  "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
  assert EDSCR.RXfull == '0';
  EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
  return;
endif

// DBGDTRRX_EL0[] (external read)  
// =========================================================================
Library pseudocode for shared/debug/dccanditr/DBGDTRTX_EL0

// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
  IMPLEMENTATION_DEFINED "generate error response";
  return bits(32) UNKNOWN;

underrun = EDSCR.Txfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
value = if underrun then bits(32) UNKNOWN else DTRTX;

if EDSCR.ERR == '1' then return value; // Error flag set: no side-effects

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then // Software lock locked: no side-effects
  return value;

if underrun then
  EDSCR.TXU = '1';  EDSCR.ERR = '1'; // Underrun condition: block side-effects
  return value;                      // Return UNKNOWN
  EDSCR.Txfull = '0';
  if Halted() && EDSCR.MA == '1' then
    EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)
    if !UsingAArch32() then
      ExecuteA64(0xB8404401<31:0>);               // A64 "LDR R1,[X0],#4"
    else
      ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
      // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
      if EDSCR.ERR == '1' then
        EDSCR.Txfull = bit UNKNOWN;
        DBGDTRTX_EL0 = bits(64) UNKNOWN;
        else
          if UsingAArch32() then
            ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
          else
            ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            assert EDSCR.Txfull == '1';
            if UsingAArch32() then
                X[1] = bits(64) UNKNOWN;
            else
                R[1] = bits(32) UNKNOWN;
            EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
          return value;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
DTRTX = value;
return;
Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
  // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
  // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
  assert N IN {32,64};
  if EDSCR.TXfull == '1' then
    value = bits(N) UNKNOWN;
  // On a 64-bit write, implement a half-duplex channel
  if N == 64 then DTRRX = value<63:32>;
  DTRTX = value<31:0>;        // 32-bit or 64-bit write
  EDSCR.TXfull = '1';
  return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
  // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
  // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
  assert N IN {32,64};
  bits(N) result;
  if EDSCR.RXfull == '0' then
    result = bits(N) UNKNOWN;
  else
    // On a 64-bit read, implement a half-duplex channel
    // NOTE: the word order is reversed on reads with regards to writes
    if N == 64 then result<63:32> = DTRTX;
    result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;

Library pseudocode for shared/debug/dccanditr/DTR

bits(32) DTRRX;
bits(32) DTRTX;
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.
EDITR[boolean memory_mapped] = bits(32) value
   if EDPRS<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
      IMPLEMENTATION_DEFINED "generate error response";
      return;
   if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write
   // The Software lock is OPTIONAL.
   if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
   if !Halted() then return;                           // Non-debug state: ignore write
   if EDSCR.ITE == '0' || EDSCR.MA == '1' then
      EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
      return;
   // ITE indicates whether the processor is ready to accept another instruction; the processor
   // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
   // is no indication that the pipeline is empty (all instructions have completed). In this
   // pseudocode, the assumption is that only one instruction can be executed at a time,
   // meaning ITE acts like "InstrCompl".
   EDSCR.ITE = '0';
      if !UsingAArch32() then
         ExecuteA64(value);
      else
         ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);
      EDSCR.ITE = '1';
   return;
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then UNDEFINED;
            else handle_el = EL1;
        when EL2
            if !HaveEL(EL2) then UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif !IsSecureEL2Enabled() && IsSecure() then UNDEFINED;
            else handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UNDEFINED;
            handle_el = EL3;
        otherwise
            Unreachable();
    from_secure = IsSecure();
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32(); // Cannot move from AArch64 to AArch32
    case handle_el of
        when EL1
            AArch32.WriteMode(M32_Svc);
            if HavePANExt() && SCTLR.SPAN == '0' then
                PSTATE.PAN = '1';
        when EL2
            AArch32.WriteMode(M32_Hyp);
        when EL3
            AArch32.WriteMode(M32_Monitor);
            if HavePANExt() then
                if !from_secure then
                    PSTATE.PAN = '0';
                    elsif SCTLR.SPAN == '0' then
                        PSTATE.PAN = '1';
                if handle_el == EL2 then
                    ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
                else
                    LR = bits(32) UNKNOWN;
                    SPSR[] = bits(32) UNKNOWN;
                    PSTATE.E = SCTLR[].EE;
                    DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
                end if
            else // Targeting AArch64
                if UsingAArch32() then
                    AArch64.MaybeZeroRegisterUppers();
                    MaybeZeroSVEUppers(target_el);
                    PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
                    if HavePANExt() && (handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
                        (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                        HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0') then
                        PSTATE.PAN = '1';
                    ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(64) UNKNOWN;  ESPR[] = bits(64) UNKNOWN;
                    DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
                    if HaveUAOExt() then PSTATE.UAO = '0';
                    if HaveMTEExt() then PSTATE.TCO = '1';
                    if HaveCSRExt() && !UsingAArch32() then PSTATE.CSR = bit UNKNOWN;
                UpdateEDSCRFields(); // Update EDSCR PE state flags
                sync_errors = HaveIESB() && SCTLR[].IESB == '1';
                if HaveDoubleFaultExt() && !UsingAArch32() then
                    sync_errors = sync_errors || (SCR_EL3.EA == '1' &&
                    SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
                // SCTLR[].IESB might be ignored in Debug state.
                Shared Pseudocode Functions
if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
else
    if sync_errors then
        SynchronizeErrors();
    return;
end

// DRPSInstruction()
// =============
// Operation of the A64 DRPS and T32 ERET instructions in Debug state
DRPSInstruction()

SynchronizeContext();

sync_errors = HaveIESB() && SCTLR[].IESB == '1';
if HaveDoubleFaultExt() && !UsingAArch32() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    end
end

bits(64) spsr = SPSR[];
SetPSTATEFromPSR(spsr);

// PSTATE.{N,Z,C,V,SS,D,A,I,F} are not observable and ignored in Debug state, so
// behave as if UNKNOWN.
if UsingAArch32() then
    PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(13) UNKNOWN;
    // In AArch32, all instructions are T32 and unconditional.
    PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
    DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
else
    PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
    if HaveCSRExt() then PSTATE.CSR = bit UNKNOWN;
end

UpdateEDSCRFields(); // Update ESCR PE state flags

return;

// Library pseudocode for shared/debug/halting/DebugHalt
constant bits(6) DebugHalt_Breakpoint = '000111';
constant bits(6) DebugHalt_EDBGRO = '010011';
constant bits(6) DebugHalt_Step_Normal = '011011';
constant bits(6) DebugHalt_Step_Exclusive = '011111';
constant bits(6) DebugHalt_OSUnlockCatch = '100011';
constant bits(6) DebugHalt_ResetCatch = '100111';
constant bits(6) DebugHalt_Watchpoint = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess = '110011';
constant bits(6) DebugHalt_ExceptionCatch = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';

// Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch
DisableITRAndResumeInstructionPrefetch();

// Library pseudocode for shared/debug/halting/ExecuteA64
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);
Library pseudocode for shared/debug/halting/ExecuteT32

// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);

Library pseudocode for shared/debug/halting/ExitDebugState

// ExitDebugState()
// ===============

ExitDebugState()
assert Halted();
SynchronizeContext();

// Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
detect that the PE has restarted.
EDSCR.STATUS = '000001'; // Signal restarting
EDESR<2:0> = '000'; // Clear any pending Halting debug events

bits(64) new_pc;
bits(64) spsr;
if UsingAArch32() then
    new_pc = ZeroExtend(DLR);
spsr = ZeroExtend(DSPSR);
else
    new_pc = DLR_EL0;
spsr = DSPSR_EL0;
// If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
if UsingAArch32() then
    SetPSTATEFromPSR(spsr<31:0>); // Can update privileged bits, even at EL0
else
    SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0
if UsingAArch32() then
    if ConstrainUnpredictable Bool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
    BranchTo(new_pc<31:0>, BranchType_DBGEXIT); // AArch32 branch
else
    // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
    if spsr<4> == '1' & ConstrainUnpredictable Bool(Unpredictable_RESTARTZEROUPPERPC) then
        new_pc<63:32> = Zeros();
    BranchTo(new_pc, BranchType_DBGEXIT); // A type of branch that is never predicted
(EDSCR.STATUS,EDPRSR.SDR) = ('000010','1'); // Atomically signal restarted
UpdateEDSCRFields(); // Stop signalling PE state
DisableITRAndResumeInstructionPrefetch();
return;
Library pseudocode for shared/debug/halting/Halt

```
// Halt()
// ======
Halt(bits(6) reason)

  // Trigger other cores to halt
  CTI_SignalEvent(CrossTriggerIn_CrossHalt);

  bits(64) preferred_restart_address = ThisInstrAddr();
  bits(32) spsr_32;
  bits(64) spsr_64;
  if UsingAArch32() then
    spsr_32 = GetPSRFromPSTATE(DebugState);
  else
    spsr_64 = GetPSRFromPSTATE(DebugState);

  if (HaveBTIE() &&
      !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive, DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
      ConstrainingUnpredictableBool(Unpredictable_ZEROBTYPE)) then
    if UsingAArch32() then
      spsr_32<11:10> = '00';
    else
      spsr_64<11:10> = '00';
  else
    DLR = preferred_restart_address<31:0>;
    DSPSR = spsr_32;
  else
    DLR_EL0 = preferred_restart_address;
    DSPSR_EL0 = spsr_64;

  EDSR.ITE = '1';
  EDSR.IO = '0';
  if IsSecure() then
    EDSR.SDD = '0';
  elsif HaveEL(EL3) then
    EDSR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
  else
    assert EDSR.SDD == '1';
  EDSR.MA = '0';

  // In Debug state:
  // * PSTATE.{SS,SSBS,D,A,I,F,T} are not observable and ignored so behave-as-if UNKNOWN.
  // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
  //     are not changed on exception entry, this function also leaves them unchanged.
  // * PSTATE.{IT,T} are ignored.
  // * PSTATE.IL is ignored and behave-as-if 0.
  // * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.
  if UsingAArch32() then
    PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
  else
    PSTATE.<SS,SSBS,D,A,I,F> = bits(6) UNKNOWN;
    PSTATE.BTYPE = '00';
  if HaveCSRExt() && !UsingAArch32() then
    PSTATE.CSR = bit UNKNOWN;
  PSTATE.IL = '0';

  StopInstructionPrefetchAndEnableITR();
  EDSR.STATUS = reason;                     // Signal entered Debug state
  UpdateEDSFRFields();                     // Update EDSR PE state flags.

  return;
```
Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```plaintext
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && ESCR.HDE == '1' && OSLR_EL1.OSLK == '0';
```

Library pseudocode for shared/debug/halting/Halted

```plaintext
// Halted()
// ========

boolean Halted()
    return !(EDCR.STATUS IN {'000001', '000010'});                     // Halted
```

Library pseudocode for shared/debug/halting/HaltingAllowed

```plaintext
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    elsif IsSecure() then
        return ExternalSecureInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/halting/Restarting

```plaintext
// Restarting()
// ============

boolean Restarting()
    return ESCR.STATUS == '000001';                                    // Restarting
```

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```plaintext
StopInstructionPrefetchAndEnableITR();
```
Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```c
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()
if !Halted() then
  EDSCR.EL = '00';
  EDSCR.NS = bit UNKNOWN;
  EDSCR.RW = '1111';
else
  EDSCR.EL = PSTATE.EL;
  EDSCR.NS = if IsSecure() then '0' else '1';
  bits(4) RW;
  RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
  if PSTATE.EL != EL0 then
    RW<0> = RW<1>;
  else
    RW<0> = if UsingAArch32() then '0' else '1';
    if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
      RW<2> = RW<1>;
    else
      RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
      if !HaveEL(EL3) then
        RW<3> = RW<2>;
      else
        RW<3> = if ELUsingAArch32(EL3) then '0' else '1';
    // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
    if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
    elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
    elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
  EDSCR.RW = RW;
return;
```

Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```c
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
// Called after an exception entry or exit, that is, such that IsSecure() and PSTATE.EL are correct
// for the exception target.
base = if IsSecure() then 0 else 4;
if HaltingAllowed() then
  if HaveExtendedECDebugEvents() then exception_exit = !exception_entry;
  ctrl = EDECCR<UInt>(PSTATE.EL) + base + 8>:EDECCR<UInt>(PSTATE.EL) + base>;
  case ctrl of
    when '00'  halt = FALSE;
    when '01'  halt = TRUE;
    when '10'  halt = (exception_exit == TRUE);
    when '11'  halt = (exception_entry == TRUE);
  else
    halt = (EDECCR<UInt>(PSTATE.EL) + base> == '1');
  if halt then Halt(DebugHalt_ExceptionCatch);
```
Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elseif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
    end

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if (HaveDoPD() && CTIDEVCTL.OSUCE == '1')
        || (!HaveDoPD() && EDECR.OSUCE == '1')
    then
        if !Halted() then EDESR.OSUC = '1';

Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);

Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
// CheckSoftwareAccessToDebugRegisters()
// -------------------------------------
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHaltSoftwareAccess);

// ExternalDebugRequest()
// ===============

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.

// Returns TRUE if the previously executed instruction was executed in the inactive state, that is, // if it was not itself stepped.
boolean HaltingStep_DidNotStep();

// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction // executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();

// RunHaltingStep()
// ============

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall, boolean reset)
    if reset then assert !Halted(); // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();
    if active && reset then // Coming out of reset with EDECR.SS set
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';
    return;
Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```java
// ExternalDebugInterruptsDisabled()
// ---------------------------------
// Determine whether EDSCR disables interrupts routed to 'target'
boolean ExternalDebugInterruptsDisabled(bits(2) target)
if Havev8p4Debug() then
    if target == EL3 || IsSecure() then
        int_dis = (EDSCR.INTdis[0] == '1' && ExternalSecureInvasiveDebugEnabled());
    else
        int_dis = (EDSCR.INTdis[0] == '1');
else
    case target of
        when EL3
            int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
        when EL2
            int_dis = (EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled());
        when EL1
            if IsSecure() then
                int_dis = (EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled());
            else
                int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
    return int_dis;
```

Library pseudocode for shared/debug/interrupts/InterruptID

```java
enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIRQ, InterruptID_COMMRX, InterruptID_COMMTX};
```

Library pseudocode for shared/debug/interrupts/SetInterruptRequestLevel

```java
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```

Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```java
// CreatePCSample()
// ================
CreatePCSample()
// In a simple sequential execution of the program, CreatePCSample is executed each time the PE
// executes an instruction that can be sampled. An implementation is not constrained such that
// reads of EDPCSRlo return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ns = if IsSecure() then '0' else '1';
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
    pc_sample.has_el2 = EL2Enabled();

    if EL2Enabled() then
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elseif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VTTBR_EL2.VMID;
        if (HaveVirtHostExt()) || HaveV82Debug() then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
        pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
```
// EDPCSRlo[] (read)
// =================
bits(32) EDPCSRlo[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
  IMPLEMENTATION_DEFINED "generate error response";
  return bits(32) UNKNOWN;
// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
  sample = pc_sample.pc<31:0>;
  if update then
    if HaveVirtHostExt() && EDSCR.SC2 == '1' then
      EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
      EDPCSRhi.EL = pc_sample.el;
      EDPCSRhi.NS = pc_sample.ns;
    else
      EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
      EDCIDSR = pc_sample.contextidr;
    if (HaveVirtHostExt() || HaveV82Debug()) && EDSCR.SC2 == '1' then
      EDVIDSR = (if HaveEL(EL2) && pc_sample.ns == '1' then pc_sample.contextidr_el2
        else bits(32) UNKNOWN);
    else
      if HaveEL(EL2) && pc_sample.ns == '1' && pc_sample.el IN {EL1, EL0} then
        EDVIDSR.VMID = pc_sample.vmid;
      else
        EDVIDSR.VMID = Zeros();
        EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
        EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
        // The conditions for setting HV are not specified if PCSRhi is zero.
        // An example implementation may be "pc_sample.rw".
        EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
      sample = Ones(32);
      if update then
        EDPCSRhi = bits(32) UNKNOWN;
        EDCIDSR = bits(32) UNKNOWN;
        EDVIDSR = bits(32) UNKNOWN;
  return sample;

Library pseudocode for shared/debug/samplebasedprofiling/PCSample

type PCSample is (boolean valid,
  bits(64) pc,
  bits(2) el,
  bit rw,
  bit ns,
  boolean has_el2,
  bits(32) contextidr,
  bits(32) contextidr_el2,
  boolean el0h,
  bits(16) vmid
)

PCSample pc_sample;
Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

// PMPCSR[] (read)
// ===============
bits(32) PMPCSR[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
  IMPLEMENTATION_DEFINED "generate error response";
  return bits(32) UNKNOWN;

// The Software lock is OPTIONAL.
update = !memory_mapped || PMLSR.SLK == '0'; // Software locked: no side-effects

if pc_sample.valid then
  sample = pc_sample.pc<31:0>;
  if update then
    PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
    PMPCSR.EL = pc_sample.el;
    PMPCSR.NS = pc_sample.ns;
    PMCID1SR = pc_sample.contextidr;
    PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;
    PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1, EL0} && !pc_sample.el0h then pc_sample.vmid else bits(16) UNKNOWN);
  else
    sample = Ones(32);
    if update then
      PMPCSR<55:32> = bits(24) UNKNOWN;
      PMPCSR.EL = bits(2) UNKNOWN;
      PMPCSR.NS = bit UNKNOWN;
      PMCID1SR = bits(32) UNKNOWN;
      PMCID2SR = bits(32) UNKNOWN;
      PMVIDSR.VMID = bits(16) UNKNOWN;

  return sample;

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

// CheckSoftwareStep()
// ===============
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

// Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
// AArch32 state. However, because Software Step is only active when the debug target Exception
// level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
step_enabled = !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() && MDSCR_EL1.SS == '1';
if step_enabled && PSTATE.SS == '0' then
  AArch64.SoftwareStepException();
Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    if UsingAArch32() then
        assert N == 32;
        assert N == 64;
    else
        assert Halted() || Restarting() || PSTATE.EL != EL0;

    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();

    if IllegalExceptionReturn(spsr) then
        dest = PSTATE.EL;
    else
        (valid, dest) = ELFromSPSR(spsr);  assert valid;

    dest_is_secure = IsSecureBelowEL3() || dest == EL3;
    dest_using_32 = (if dest == EL0 then spsr<4> == '1' else ELUsingAArch32(dest));
    if dest_using_32 then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, dest_is_secure);
    else
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, dest_is_secure, mask);

    ELd = DebugTargetFrom(dest_is_secure);
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';

    return SS_bit;

Library pseudocode for shared/debug/softwarestep/SSAdvance

// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is, // if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB or ERET executed // in the active-not-pending state, or if another exception was taken before the Software Step exception. // Returns FALSE otherwise, indicating that the previously executed instruction was executed in the // active-not-pending state, that is, the instruction was stepped.
boolean SoftwareStep_DidNotStep();

// Returns a value that describes the previously executed instruction. The result is valid only if // SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX that failed its condition code test.
// Otherwise returns TRUE if the instruction was a Load-Exclusive class instruction, and FALSE if the // instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();

bits(5) ConditionSyndrome()
// ===============
// Return CV and COND fields of instruction syndrome

bits(5) syndrome;
if UsingAArch32() then
    cond = AArch32.CurrentCond();
    if PSTATE.T == '0' then  // A32
        syndrome<4> = '1';
        // A conditional A32 instruction that is known to pass its condition code check // can be presented either with COND set to 0xE, the value for unconditional, or // the COND value held in the instruction.
        if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
            syndrome<3:0> = '1110';
        else
            syndrome<3:0> = cond;
    else  // T32
        // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether: // * CV set to 0 and COND is set to an UNKNOWN value // * CV set to 1 and COND is set to the condition code for the condition that // applied to the instruction.
        if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
            syndrome<4> = '1';
            syndrome<3:0> = cond;
        else
            syndrome<4> = '0';
            syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';
else
    syndrome<4> = '1';
    syndrome<3:0> = '1110';
return syndrome;
Library pseudocode for shared/exceptions/exceptions/Exception

```c
enumeration Exception {
    Exception_Uncategorized, // Uncategorized or unknown reason
    Exception_WFxTrap,       // Trapped WFI or WFE instruction
    Exception_CP15RTTrap,    // Trapped AArch32 MCR or MRC access to CP15
    Exception_CP15RTTRap,    // Trapped AArch32 MCR or MRC access to CP15
    Exception_CP14RTTrap,    // Trapped AArch32 MCR or MRC access to CP14
    Exception_CP14DTRap,     // Trapped AArch32 LDC or STC access to CP14
    Exception_AdvSIMDFPAccessTrap, // HCPTTR-trapped access to SIMD or FP
    Exception_FPIDTrap,      // Trapped access to SIMD or FP ID register
    Exception_LDST64BTrap,   // Trapped access to ST64BV, ST64BV0, ST64B and LD64B
    // Trapped BXJ instruction not supported in Armv8
    Exception_PKTTrap,       // Trapped invalid PAC use
    Exception_CP14RRTrap,    // Trapped MRRC access to CP14 from AArch32
    Exception_IllegalState, // Illegal Execution state
    Exception_SupervisorCall,// Supervisor Call
    Exception_HypervisorCall,// Hypervisor Call
    Exception_MonitorCall,   // Monitor Call or Trapped SMC instruction
    Exception_SystemRegisterTrap, // Trapped MRS or MSR system register access
    Exception_ERetTrap,      // Trapped invalid ERET use
    Exception_InstructionAbort, // Instruction Abort or Prefetch Abort
    Exception_PCAAlignment,  // PC alignment fault
    Exception_DataAbort,     // Data Abort
    Exception_NV2DataAbort,  // Data abort at EL1 reported as being from EL2
    Exception_PACFail,      // PAC Authentication failure
    Exception_SPAlignment,   // SP alignment fault
    Exception_FPTrappedException, // IEEE trapped FP exception
    Exception_SError,       // SError interrupt
    Exception_Breakpoint,  // (Hardware) Breakpoint
    Exception_SoftwaresStep, // Software Step
    Exception_Watchpoint,   // Watchpoint
    Exception_NV2Watchpoint, // Watchpoint at EL1 reported as being from EL2
    Exception_SoftwareBreakpoint, // Software Breakpoint Instruction
    Exception_VectorCatch,  // AArch32 Vector Catch
    Exception_IRQ,          // IRQ interrupt
    Exception_SVEAccessTrap, // HCPTTR trapped access to SVE
    Exception_TSTARTAccessTrap, // Trapped TSTART access
    Exception_BranchTarget, // Branch Target Identification
    Exception_FIQ};         // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```c
library pseudocode for shared/exceptions/exceptions/ExceptionRecord

type ExceptionRecord is (Exception exceptype, // Exception class
    bits(25) syndrome,       // Syndrome record
    bits(5) syndrome2,       // Syndrome record
    bits(64) vaddress,       // Virtual fault address
    boolean ipavalid,        // Physical fault address for second stage faults in
    bits(1) NS,              // Physical fault address for second stage faults in
    bits(52) ipaddress)      // Physical fault address for second stage faults in
```
Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```c
// ExceptionSyndrome()
// ================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)
{
    ExceptionRecord r;
    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.syndrome2 = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros();

    return r;
}
```

Library pseudocode for shared/exceptions/traps/ReservedValue

```c
// ReservedValue()
// ===============

ReservedValue()
{
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();
}
```

Library pseudocode for shared/exceptions/traps/UnallocatedEncoding

```c
// UnallocatedEncoding()
// =====================

UnallocatedEncoding()
{
    if UsingAArch32() && AArch32.ExecutingCP10or11Instr() then
        FPEXC.DEX = '0';
    else
        AArch64.TakeUndefInstrException();
}
```
Library pseudocode for shared/functions/aborts/EncodeLDFSC

// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)

    bits(6) result;
    case statuscode of
        when Fault_AddressSize result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission result = '0011':level<1:0>; assert level IN {1,2,3};
        when Fault_Translation result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal result = '010000';
        when Fault_SyncExternalOnWalk result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity result = '011000';
        when Fault_SyncParityOnWalk result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity result = '011001';
        when Fault_AsyncExternal result = '010001';
        when Fault_AsyncParity result = '011001';
        when Fault_AccessFlag result = '010001';
        when Fault_Translation result = '100001';
        when Fault_HWUpdateAccessFlag result = '110001';
        when Fault_Lockdown result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive result = '110101';  // IMPLEMENTATION DEFINED
        otherwise unreachable();
    return result;

Library pseudocode for shared/functions/aborts/IPAValid

// IPAValid()
// ===========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
assert fault.statuscode != Fault_None;

    if fault.s2fs1walk then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Permission, Fault_Translation, Fault_AddressSize};
    elsif fault.secondstage then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;

Library pseudocode for shared/functions/aborts/IsAsyncAbort

// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
Library pseudocode for shared/functions/aborts/IsDebugException

// IsDebugException()
// ===============

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;

Library pseudocode for shared/functions/aborts/IsExternalAbort

// IsExternalAbort()
// ===============

// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk, Fault_AsyncExternal, Fault_AsyncParity });

// IsExternalAbort()
// ===============

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode);

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

// IsExternalSyncAbort()
// ===============

// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk });

// IsExternalSyncAbort()
// ===============

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode);

Library pseudocode for shared/functions/aborts/IsFault

// IsFault()
// ===========

// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;
Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

// IsSErrorInterrupt()
// ===============
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;
    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===============

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);

Library pseudocode for shared/functions/aborts/IsSecondStage

// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.secondstage;

Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

// Returns the extended syndrome information for a second stage fault.
// <10> - Syndrome valid bit. The syndrome is only valid for certain types of access instruction.
// <9:8> - Access size.
// <7> - Sign extended (for loads).
// <6:2> - Transfer register.
// <1> - Transfer register is 64-bit.
// <0> - Instruction has acquire/release semantics.

bits(11) LSInstructionSyndrome();

Library pseudocode for shared/functions/common/ASR

// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;

Library pseudocode for shared/functions/common/ASR_C

// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<<shift+N-shift-1;
    carry_out = extended_x<<shift-1;
    return (result, carry_out);
Library pseudocode for shared/functions/common/Abs

```c
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;
// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

Library pseudocode for shared/functions/common/Align

```c
// Align()
// ========

integer Align(integer x, integer y)
    return y * (x DIV y);
// Align()
// ========

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

Library pseudocode for shared/functions/common/BitCount

```c
// BitCount()
// =========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

Library pseudocode for shared/functions/common/CountLeadingSignBits

```c
// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

Library pseudocode for shared/functions/common/CountLeadingZeroBits

```c
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```
Library pseudocode for shared/functions/common/Elem

// Elem[] - non-assignment form
// ============================

bits(size) Elem<bits(N) vector, integer e, integer size]
   assert e >= 0 && (e+1)*size <= N;
   return <e+size : e*size>;

// Elem[] - assignment form
// ========================

Elem<bits(N) &vector, integer e] = bits(size) value
   assert e >= 0 && (e+1)*size <= N;
   vector<e+size : e*size> = value;
   return;

Library pseudocode for shared/functions/common/Extend

// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
   return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// ========

bits(N) Extend(bits(M) x, boolean unsigned)
   return Extend(x, N, unsigned);

Library pseudocode for shared/functions/common/HighestSetBit

// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
   for i = N downto 0
      if x<i> == '1' then return i;
   return -1;

Library pseudocode for shared/functions/common/Int

// Int()
// =====

integer Int<bits(N) x, boolean unsigned)
   result = if unsigned then UInt(x) else SInt(x);
   return result;
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);

// IsZero()
// =======

boolean IsZero(bits(N) x)
    return x == Zeros(N);

// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';

// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;

// LSL_C()
// =======

<bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);

// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
Library pseudocode for shared/functions/common/LSR_C

// LSR_C()
// ========
(bits(N), bit) LSR_C(bits(N) x, integer shift)
  assert shift > 0;
  extended x = ZeroExtend(x, shift+N);
  result = extended x<shift+N:shift>;
  carry_out = extended x<shift-1>;
  return (result, carry_out);

Library pseudocode for shared/functions/common/LowestSetBit

// LowestSetBit()
// =============
integer LowestSetBit(bits(N) x)
  for i = 0 to N-1
    if x<i> == '1' then return i;
  return N;

Library pseudocode for shared/functions/common/Max

// Max()
// ======
integer Max(integer a, integer b)
  return if a >= b then a else b;

// Max()
// ======
real Max(real a, real b)
  return if a >= b then a else b;

Library pseudocode for shared/functions/common/Min

// Min()
// ======
integer Min(integer a, integer b)
  return if a <= b then a else b;

// Min()
// ======
real Min(real a, real b)
  return if a <= b then a else b;

Library pseudocode for shared/functions/common/Ones

// Ones()
// ======
bits(N) Ones(integer N)
  return Replicate('1',N);

// Ones()
// ======
bits(N) Ones()
  return Ones(N);
Library pseudocode for shared/functions/common/ROR

// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
assert shift >= 0;
if shift == 0 then
  result = x;
else
  (result, -) = ROR_C(x, shift);
return result;

Library pseudocode for shared/functions/common/ROR_C

// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
assert shift != 0;
m = shift MOD N;
result = LSR(x, m) OR LSL(x, N-m);
carry_out = result<N-1>;
return (result, carry_out);

Library pseudocode for shared/functions/common/Replicate

// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
assert N MOD M == 0;
return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);

Library pseudocode for shared/functions/common/RoundDown

integer RoundDown(real x);

Library pseudocode for shared/functions/common/RoundTowardsZero

// RoundTowardsZero()
// ================

integer RoundTowardsZero(real x)
return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);

Library pseudocode for shared/functions/common/RoundUp

integer RoundUp(real x);

Library pseudocode for shared/functions/common/SInt

// SInt()
// =====

integer SInt(bits(N) x)
result = 0;
for i = 0 to N-1
  if x<i> == '1' then result = result + 2^i;
  if x<N-1> == '1' then result = result - 2^N;
return result;
Library pseudocode for shared/functions/common/SignExtend

// SignExtend()
// ============
bits(N) SignExtend(bits(M) x, integer N)
   assert N >= M;
   return Replicate(x<N-M>, N-M) : x;
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x)
   return SignExtend(x, N);

Library pseudocode for shared/functions/common/UInt

// UInt()
// ======
integer UInt(bits(N) x)
   result = 0;
   for i = 0 to N-1
      if x<i> == '1' then result = result + 2^i;
   return result;

Library pseudocode for shared/functions/common/ZeroExtend

// ZeroExtend()
// ============
bits(N) ZeroExtend(bits(M) x, integer N)
   assert N >= M;
   return Zeros(N-M) : x;
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x)
   return ZeroExtend(x, N);

Library pseudocode for shared/functions/common/Zeros

// Zeros()
// =======
bits(N) Zeros(integer N)
   return Replicate('0',N);
// Zeros()
// =======

bits(N) Zeros()
   return Zeros(N);

Library pseudocode for shared/functions/crc/BitReverse

// BitReverse()
// ===========
bits(N) BitReverse(bits(N) data)
   bits(N) result;
   for i = 0 to N-1
      result<N-i-1> = data<i>;
   return result;
Library pseudocode for shared/functions/crc/HaveCRCExt

```c
// HaveCRCExt()
// ============

boolean HaveCRCExt()
return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```c
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
assert N > 32;
for i = N-1 downto 32
    if data<i> == '1' then
        data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
return data<31:0>;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```c
// AESInvMixColumns()
// =================

// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR FFmul09(in3<c*8+:8>);
        out1<c*8+:8> = FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR FFmul0D(in3<c*8+:8>);
        out2<c*8+:8> = FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR FFmul0E(in3<c*8+:8>);
        out3<c*8+:8> = FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR FFmul0E(in3<c*8+:8>);

    return (out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8>:
             out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8>:
             out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8>:
             out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>);
```

Library pseudocode for shared/functions/crypto/AESInvShiftRows

```c
// AESInvShiftRows()
// =================

// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
return (op< 24+:8> : op< 48+:8> : op< 72+:8> : op< 96+:8>:
         op<120+:8> : op< 16+:8> : op< 40+:8> : op< 64+:8>:
         op< 88+:8> : op<112+:8> : op<  8+:8> : op< 32+:8>:
         op< 56+:8> : op< 80+:8> : op<104+:8> : op<  0+:8>);
```
Library pseudocode for shared/functions/crypto/AESInvSubBytes

// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    bits(16*16*8) GF2_inv = { /* Inverse S-box values */
        /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0>:
        /*E*/ 0x61995383cbbebc8b0f52ae4d3be0a0<127:0>:
        /*D*/ 0xef9cc939f7ae52d04ab519a97f5160<127:0>:
        /*C*/ 0x5f6ec0827591012b131c7078833a8dd1f<127:0>:
        /*B*/ 0xf45ac078feco0b9a2079d2c64b3e56fc<127:0>:
        /*A*/ 0x11be18aa062b76f89c5291d71af147<127:0>:
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0>:
        /*8*/ 0x73e6b400cecf297eadc67424f111913a<127:0>:
        /*7*/ 0x6b8a130102da0f81e2cd0<127:0>:
        /*6*/ 0x645b3b705l8e4f70ad3bce0abcd90<127:0>:
        /*5*/ 0x849d8d47546155edab9edf0504876c<127:0>:
        /*4*/ 0x926655d6ca4d41698686466f872<127:0>:
        /*3*/ 0x2512d67b242d28631e08<127:0>:
        /*2*/ 0x4ec3fa420954ceed23ca632947b54<127:0>:
        /*1*/ 0x0be9dec444438e347ff29b8239e37c<127:0>:
        /*0*/ 0xfb7f3819ea340bf38a53630d56a9052<127:0>;
    }
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>*8+:8)>;
    return out;

Library pseudocode for shared/functions/crypto/AESMixColumns

// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the State and mixes their data (independently of one another) to produce new columns.

bits(128) AESMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<0*c8+:8> = FFMul02(in0<0*c8+:8>) EOR FFMul03(in1<0*c8+:8>) EOR in2<0*c8+:8> EOR in3<0*c8+:8> EOR
        out1<0*c8+:8> = in0<0*c8+:8> EOR FFMul02(in1<0*c8+:8>) EOR FFMul03(in2<0*c8+:8>) EOR
        out2<0*c8+:8> = in0<0*c8+:8> EOR in1<0*c8+:8> EOR FFMul02(in2<0*c8+:8>) EOR FFMul03(in3<0*c8+:8>) EOR
        out3<0*c8+:8> = FFMul03(in0<0*c8+:8>) EOR in1<0*c8+:8> EOR in2<0*c8+:8> EOR FFMul02(in3<0*c8+:8>) EOR

    return {
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>:
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8>:
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8>:
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    };

Shared Pseudocode Functions
Library pseudocode for shared/functions/crypto/AESShiftRows

```c
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
return {
    op< 88+:8> : op< 48+:8> : op<  8+:8> : op< 96+:8> :
    op< 56+:8> : op<104+:8> : op< 64+:8> :
    op< 24+:8> : op<112+:8> : op< 72+:8> : op< 32+:8> :
    op<120+:8> : op< 80+:8> : op< 40+:8> : op<  0+:8>
};
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```c
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
// S-box values
bits(16*16*8) GF2 = {
    0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
    0xdf2855cee9871e9b948ed969119f8e1<127:0> :
    0x9e1dc186b95735610f603486b53e70<127:0> :
    0x8a8bbd4b1f74de8c6b4a61c2e2578ba<127:0> :
    0x08ae7a65ea4566ca94ed58d6d37c8e7<127:0> :
    0x79e4959162ac3c25c2406490a3a32e0<127:0> :
    0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
    0x73195d643d7ea7c41744975fec130ccdd<127:0> :
    0x2f3f1021dab6bfc5389d928f40a351<127:0> :
    0xa8f3c07f0f9458533ad43fbaae6f0<127:0> :
    0xcf5f8c4a39be6a5bb1f20ed00d153<127:0> :
    0x842fe329b3d63b52a05a6f1ba1c8309<127:0> :
    0x75b227ebe28012079a859618c323c704<127:0> :
    0x1531d871fe5a534ccf73f362693f9db7<127:0> :
    0xc872a49cafa2d4afa94759fa7dc982ca<127:0> :
    0x76ab77fe2b670130c56f6bf27b777c63<127:0>
};
bits(128) out;
for i = 0 to 15
    out<i*8+:8> = GF2<UInt>(op<i*8+:8>)*8+:8>
return out;
```
Library pseudocode for shared/functions/crypto/FFmul02
// FFmul02()
// =========
bits(8) FFmul02(bits(8) b)
bits(256*8) FFmul_02 = (
/*
F E D C B A 9 8 7 6 5 4 3 2 1 0
*/
/*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
/*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
/*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
/*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
/*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
/*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
/*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
/*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
/*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
/*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
/*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
/*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
/*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
/*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
/*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
/*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
);
return FFmul_02<UInt(b)*8+:8>;

Library pseudocode for shared/functions/crypto/FFmul03
// FFmul03()
// =========
bits(8) FFmul03(bits(8) b)
bits(256*8) FFmul_03 = (
/*
F E D C B A 9 8 7 6 5 4 3 2 1 0
*/
/*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
/*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
/*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
/*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
/*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
/*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
/*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
/*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
/*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
/*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
/*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
/*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
/*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
/*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
/*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
/*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
);
return FFmul_03<UInt(b)*8+:8>;

Shared Pseudocode Functions

Page 3307


Library pseudocode for shared/functions/crypto/FFmul09

```plaintext
// FFmul09()
// =========
bits(8) FFmul09(bits(8) b)
bits(256*8) FFmul_09 = ( /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
    /* F */ 0x464F545D626B70790E071C152A233831<127:0> : /* F */
    /* E */ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> : /* E */
    /* D */ 0x7D746F6659504B42353C2721118030A<127:0> : /* D */
    /* C */ 0x0DE4FF66C900DB2A5ACB7BE81B8939A<127:0> : /* C */
    /* B */ 0x3039222B141D060F78716A635C54E47<127:0> : /* B */
    /* A */ 0xA0A9B2BB848D969FE8E1FAF3CC5DED7<127:0> : /* A */
    /* 9 */ 0x0B0219102F2634434A5158676E757C<127:0> : /* 9 */
    /* 8 */ 0xB98B9808FB6ADA4D3AC1B0F7EE5E9C<127:0> : /* 8 */
    /* 7 */ 0xAAA3BB818E79C95E2EBF09C6CDF4DD<127:0> : /* 7 */
    /* 6 */ 0x3332B211E170C5727B60956F544D<127:0> : /* 6 */
    /* 5 */ 0x9198B3A85C7332D76443A5158676E757C<127:0> : /* 5 */
    /* 4 */ 0x0108131A252C373E49405B526D647F67<127:0> : /* 4 */
    /* 3 */ 0xDD5CE7F813AE9490868FB90A2AB<127:0> : /* 3 */
    /* 2 */ 0x4C58E768617A30401D6F209323B<127:0> : /* 2 */
    /* 1 */ 0x7E6F5FCC3ACD1D8AF6DBD8B829999<127:0> : /* 1 */
    /* 0 */ 0x777E656C535A41483F362D241B120900<127:0> : /* 0 */
    );
return FFmul_09<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0B

```plaintext
// FFmul0B()
// =========
bits(8) FFmul0B(bits(8) b)
bits(256*8) FFmul_0B = ( /* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
    /* F */ 0xA3A8B5BE8F849992FB0E6ED70CC1A<127:0> : /* F */
    /* E */ 0x1318050E3F342224B405D56676C717A<127:0> : /* E */
    /* D */ 0x2D38E754F4E898B9690AC7A7AB1<127:0> : /* D */
    /* C */ 0x8863775444F5239332B26DC170A01<127:0> : /* C */
    /* B */ 0x555E434879726F480D61B12122A373C<127:0> : /* B */
    /* A */ 0xE5EFE7F8C92F0D4B06A0A919A878C<127:0> : /* A */
    /* 9 */ 0x2E253B330209141F767066B5A514C47<127:0> : /* 9 */
    /* 8 */ 0x9E69B8B282894A4F6C62008BEAE1FCF7<127:0> : /* 8 */
    /* 7 */ 0x545F42A478736E5650C71A1228363D<127:0> : /* 7 */
    /* 6 */ 0xE4E2F9C83D65BC7AA19098668D<127:0> : /* 6 */
    /* 5 */ 0x2F243932030811E777G16A5894046<127:0> : /* 5 */
    /* 4 */ 0x9F94B82BB8A5AEC7C5D1DABE9FDF6<127:0> : /* 4 */
    /* 3 */ 0xA29B4B8E58993F4F1CE76D0DC0B<127:0> : /* 3 */
    /* 2 */ 0x1219040F3E3528234A14C57666D070B<127:0> : /* 2 */
    /* 1 */ 0x9D20CFC45F3EEB81A979CAD46B80<127:0> : /* 1 */
    /* 0 */ 0x69627F4454E5358313A2721D100B00<127:0> : /* 0 */
    );
return FFmul_0B<UInt(b)*8+:8>;
```
Library pseudocode for shared/functions/crypto/FFmul0D

// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
bits(256*8) FFmul_0D = (/* F E D C B A 9 8 7 6 5 4 3 2 1 0 */)
    /*F*/ 0x979A8D80A3AEB9B4FFF2E578CBD6D1DC<127:0> :
    /*E*/ 0x47A45D0737E69642F2235381B16010C<127:0> :
    /*D*/ 0x2C21363B1815020F44495E5370766C61<127:0> :
    /*C*/ 0x4CF16E8CB8C5D2DF9499E83A0D8B7<127:0> :
    /*B*/ 0xF8F7E0EDECE3D499298F85A6A8CB1<127:0> :
    /*A*/ 0x2A27303D1E130409424F5B55768C61<127:0> :
    /*9*/ 0x414C5B567578F622924333E1D10070A<127:0> :
    /*8*/ 0x919CBB86A5A88FBF29F4E3ECD0D7D7<127:0> :
    /*7*/ 0x4D04575A797463E25283F3111C0806<127:0> :
    /*6*/ 0x9D90878AA9A4B3EF58FEF21CCD66<127:0> :
    /*5*/ 0xF6FBECE12CD8D9E9A489B700B<127:0> :
    /*4*/ 0x26263C3B990805468435547A77605D<127:0> :
    /*3*/ 0x2026A3714190E03484525F7C716668<127:0> :
    /*2*/ 0xF0FDEA7C4C90D3989582F8CA1B6D8<127:0> :
    /*1*/ 0xF9B6B8CAFA2B5BF3FEE9E47CADDD<127:0> :
    /*0*/ 0x4B46515C72656823E393471A000<127:0> );
return FFmul_0D<UInt(b)*8+:8>;

Library pseudocode for shared/functions/crypto/FFmul0E

// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
bits(256*8) FFmul_0E = (/* F E D C B A 9 8 7 6 5 4 3 2 1 0 */)
    /*F*/ 0x8D83919FB5BBA9A7DF3E1FC5CBD9D7<127:0> :
    /*E*/ 0x6D6371F55549471D13010F25283937<127:0> :
    /*D*/ 0x5658A4466727C26283A341E10020C<127:0> :
    /*C*/ 0x86B8AAA8E89929CC6C8D4A4FE092EC<127:0> :
    /*B*/ 0x26263C31112908054E345545977606D<127:0> :
    /*A*/ 0x2026A3714190E03484525F7C716668<127:0> :
    /*9*/ 0xF8FBECE12CD8D9E9A489B700B<127:0> :
    /*8*/ 0x1B1506090805468435547A77605D<127:0> :
    /*7*/ 0x2C22303E141A00065C5240646467876<127:0> :
    /*6*/ 0x17190B052F213333D6767B755F151434<127:0> :
    /*5*/ 0xF7F9E9FBE5FC1D3D07899B95F6B1A3AD<127:0> :
    /*4*/ 0x616F7D7359574541111F0DD29273535<127:0> :
    /*3*/ 0x81F9D938B7A5ABF1FFEDE3C97D50B<127:0> :
    /*2*/ 0x1B445882B8C9910C4D608F2FCEEE0<127:0> :
    /*1*/ 0x5A544648626CE702A243638121C0E00<127:0> );
return FFmul_0E<UInt(b)*8+:8>;

Library pseudocode for shared/functions/crypto/HaveAESExt

// HaveAESExt()
// =========

// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

// HaveBit128PMULLExt()
// ==============
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";

Library pseudocode for shared/functions/crypto/HaveSHA1Ext

// HaveSHA1Ext()
// =============
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";

Library pseudocode for shared/functions/crypto/HaveSHA256Ext

// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";

Library pseudocode for shared/functions/crypto/HaveSHA3Ext

// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
if !HasArchVersion(ARMv8p2) || !HaveSHA1Ext() && HaveSHA256Ext() then
return FALSE;
return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";

Library pseudocode for shared/functions/crypto/HaveSHA512Ext

// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
if !HasArchVersion(ARMv8p2) || !HaveSHA1Ext() && HaveSHA256Ext() then
return FALSE;
return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
if !HasArchVersion(ARMv8p2) then
    return FALSE;
return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";

// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
if !HasArchVersion(ARMv8p2) then
    return FALSE;
return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";

// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 & shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);

// SHA256hash()
// ============

bits(128) SHA256hash(bits(128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;
    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
    <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);

// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
Library pseudocode for shared/functions/crypto/SHAHashSIGMA0

// SHAHashSIGMA0()
// ===============

bits(32) SHAHashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);

Library pseudocode for shared/functions/crypto/SHAHashSIGMA1

// SHAHashSIGMA1()
// ===============

bits(32) SHAHashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);

Library pseudocode for shared/functions/crypto/SHAmajority

// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));

Library pseudocode for shared/functions/crypto/SHAparity

// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);

Library pseudocode for shared/functions/crypto/Sbox

// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = 0xd690e9fece13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491e84f7;
    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8+8>;
    return sboxout;

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
// ClearExclusiveMonitors()
// ========================
// Clear the local Exclusives monitor for the executing PE.
ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());

// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();

// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);

// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);

// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);

// Return the ID of the currently executing PE.
integer ProcessorID();

// AArch32.HaveHPDExt()
// ================
boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);

// AArch64.HaveHPDExt()
// ================
boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
// Have52BitIPAAndPASpaceExt()
// =================
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
{
    return HasArchVersion(ARMv8p7) &&
        boolean IMPLEMENTATION_DEFINED "Has 52-bit IPA and PA support" &&
        Have52BitVAExt() && Have52BitPAExt();
}

// Have52BitPAExt()
// ================
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
{
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support";
}

// Have52BitVAExt()
// ================
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
{
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support";
}

// HaveAArch32BF16Ext()
// ====================
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
{
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension";
}

// HaveAArch32Int8MatMulExt()
// ===========================
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
{
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul extension";
}

// HaveAltFP()
// ===========
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.

boolean HaveAltFP()
{
    return HasArchVersion(ARMv8p7);
// HaveAtomicExt()
// ===============
boolean HaveAtomicExt()
return HasArchVersion(ARMv8p1);

// HaveBF16Ext()
// =============
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.
boolean HaveBF16Ext()
return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch64 BFloat16 extension");

// HaveBTIExt()
// ============
// Returns TRUE if support for Branch Target Indentification is implemented.
boolean HaveBTIExt()
return HasArchVersion(ARMv8p5);

// HaveBlockBBM()
// ==============
// Returns TRUE if support for changing block size without requiring break-before-make is implemented.
boolean HaveBlockBBM()
return HasArchVersion(ARMv8p4);

// HaveCSRExt()
// ============
// Returns TRUE if Call Stack Recorder Extension is implemented, and FALSE otherwise.
boolean HaveCSRExt()
return boolean IMPLEMENTATION_DEFINED "Has Call Stack Recorder Extension";

// HaveCommonNotPrivateTransExt()
// ==============================
boolean HaveCommonNotPrivateTransExt()
return HasArchVersion(ARMv8p2);

// HaveDGHExt()
// ============
// Returns TRUE if Data Gathering Hint instruction support is implemented, and FALSE otherwise.
boolean HaveDGHExt()
return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";
// HaveDITExt()
// ============
// boolean HaveDITExt()
// return HasArchVersion(ARMv8p4);

// HaveDOTPExt()
// =============
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.
// boolean HaveDOTPExt()
// return HasArchVersion(ARMv8p4) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has Dot Product extension");

// HaveDoPD()
// ===========
// Returns TRUE if Debug Over Power Down extension support is implemented and FALSE otherwise.
// boolean HaveDoPD()
// return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";

// HaveDoubleFaultExt()
// =====================
// boolean HaveDoubleFaultExt()
// return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB());

// HaveDoubleLock()
// ================
// Returns TRUE if support for the OS Double Lock is implemented.
// boolean HaveDoubleLock()
// return !HasArchVersion(ARMv8p4) || boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";

// HaveE0PDExt()
// =============
// Returns TRUE if support for constant fault times for unprivileged accesses to the memory map is implemented.
// boolean HaveE0PDExt()
// return HasArchVersion(ARMv8p5);

// HaveECVExt()
// ============
// Returns TRUE if Enhanced Counter Virtualization extension support is implemented, and FALSE otherwise.
// boolean HaveECVExt()
// return HasArchVersion(ARMv8p6);
Library pseudocode for shared/functions/extension/HaveEMPAMExt

```cpp
boolean HaveEMPAMExt()
{
    return HasArchVersion(ARMv8p6) && HaveEMPAMExt() &&
           boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension";
}
```

Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```cpp
boolean HaveExtendedCacheSets()
{
    return HasArchVersion(ARMv8p3);
}
```

Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```cpp
boolean HaveExtendedECDebugEvents()
{
    return HasArchVersion(ARMv8p2);
}
```

Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```cpp
boolean HaveExtendedExecuteNeverExt()
{
    return HasArchVersion(ARMv8p2);
}
```

Library pseudocode for shared/functions/extension/HaveFCADDExt

```cpp
boolean HaveFCADDExt()
{
    return HasArchVersion(ARMv8p3);
}
```

Library pseudocode for shared/functions/extension/HaveFGTExt

```cpp
boolean HaveFGTExt()
{
    return HasArchVersion(ARMv8p6) && !ELUsingAArch32(EL2);
}
```

Library pseudocode for shared/functions/extension/HaveFJCVTZSExt

```cpp
boolean HaveFJCVTZSExt()
{
    return HasArchVersion(ARMv8p3);
}
```
Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

// HaveFP16MulNoRoundingToFP32Ext()
// ================================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions,
// and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
if !HaveFP16Ext() then return FALSE;
if HasArchVersion(ARMv8p4) then return TRUE;
return (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");

Library pseudocode for shared/functions/extension/HaveFeatLS64

// HaveFeatLS64()
// =-------------
// Returns TRUE if the LD64B, ST64B, ST64BV, and ST64BV0 instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
return (HasArchVersion(ARMv8p7) && boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");

Library pseudocode for shared/functions/extension/HaveFeatRPRES

// HaveFeatRPRES()
// =---------------
// Returns TRUE if the Reciprocal Estimate and Reciprocal Square Root Estimate
// instructions have increased precision, and FALSE otherwise.

boolean HaveFeatRPRES()
return (HasArchVersion(ARMv8p7) && (boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate precision support") && HaveAltFP());

Library pseudocode for shared/functions/extension/HaveFeatWFxT

// HaveFeatWFxT()
// =-------------
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatWFxT()
return HasArchVersion(ARMv8p7);

Library pseudocode for shared/functions/extension/HaveFeatXS

// HaveFeatXS()
// =-----------
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
// are supported, and FALSE otherwise.

boolean HaveFeatXS()
return HasArchVersion(ARMv8p7);

Library pseudocode for shared/functions/extension/HaveFlagFormatExt

// HaveFlagFormatExt()
// ===================
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFlagFormatExt()
return HasArchVersion(ARMv8p5);
// HaveFlagManipulateExt()
// =======================
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
    return HasArchVersion(ARMv8p4);

// HaveFrintExt()
// ===============
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()
    return HasArchVersion(ARMv8p5);

// HaveHCRXEL2Ext()
// ================
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveHCRXEL2Ext()
    return HasArchVersion(ARMv8p7);

// HaveHPMDEExt()
// =============

boolean HaveHPMDEExt()
    return HasArchVersion(ARMv8p1);

// HaveIDSExt()
// ===========

boolean HaveIDSExt()
    return HasArchVersion(ARMv8p4);

// HaveIESB()
// =========

boolean HaveIESB()
    return (HaveRASExt() &&
            boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");

// HaveInt8MatMulExt()
// ===================
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch64 Int8 Mat Mul extension");
Library pseudocode for shared/functions/extension/HaveLSE2Ext

// HaveLSE2Ext()
// =============
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.
boolean HaveLSE2Ext()
    return HasArchVersion(ARMv8p4);

Library pseudocode for shared/functions/extension/HaveMPAMExt

// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.
boolean HaveMPAMExt()
    return (HasArchVersion(ARMv8p2) &
            boolean IMPLEMENTATION_DEFINED "Has MPAM extension");

Library pseudocode for shared/functions/extension/HaveMTE2Ext

// HaveMTE2Ext()
// =============
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.
boolean HaveMTE2Ext()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension";

Library pseudocode for shared/functions/extension/HaveMTE3Ext

// HaveMTE3Ext()
// =============
// Returns TRUE if MTE Asymmetric Fault Handling support is
// implemented, and FALSE otherwise.
boolean HaveMTE3Ext()
    return ((HasArchVersion(ARMv8p7) &
             HaveMTE2Ext()) || (HasArchVersion(ARMv8p5) &
                          boolean IMPLEMENTATION_DEFINED "Has MTE3 extension");

Library pseudocode for shared/functions/extension/HaveMTEExt

// HaveMTEExt()
// ===========
// Returns TRUE if MTE implemented, and FALSE otherwise.
boolean HaveMTEExt()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";

Library pseudocode for shared/functions/extension/HaveNV2Ext

// HaveNV2Ext()
// =============
// Returns TRUE if Enhanced Nested Virtualization is implemented.
boolean HaveNV2Ext()
    return (HasArchVersion(ARMv8p4) &
            HaveNVExt()
            &
            boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");
// HaveNVExt()
// ===========
// Returns TRUE if Nested Virtualization is implemented.

boolean HaveNVExt()
{
    return HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization";
}

// HaveNoSecurePMUDisableOverride()
// ================================

boolean HaveNoSecurePMUDisableOverride()
{
    return HasArchVersion(ARMv8p2);
}

// HaveNoninvasiveDebugAuth()
// ==========================
// Returns TRUE if the Non-invasive debug controls are implemented.

boolean HaveNoninvasiveDebugAuth()
{
    return !HasArchVersion(ARMv8p4);
}

// HavePAN3Ext()
// =============
// Returns TRUE if SCTLR_EL1.EPAN and SCTLR_EL2.EPAN support is implemented,
// and FALSE otherwise.

boolean HavePAN3Ext()
{
    return HasArchVersion(ARMv8p7) || (HasArchVersion(ARMv8p1) &&
    boolean IMPLEMENTATION_DEFINED "Has PAN3 extension");
}

// HavePANExt()
// ============

boolean HavePANExt()
{
    return HasArchVersion(ARMv8p1);
}

// HavePMUv3p7()
// =============
// Returns TRUE if the PMUv3p7 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p7()
{
    return (HasArchVersion(ARMv8p7) && Havev85PMU() &&
    boolean IMPLEMENTATION_DEFINED "Has PMUv3p7 extension");
}

// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
{
    return HasArchVersion(ARMv8p2);
}
Library pseudocode for shared/functions/extension/HavePrivATExt

// HavePrivATExt()
// ===============

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

Library pseudocode for shared/functions/extension/HaveAccessFlagUpdateExt()

// HaveAccessFlagUpdateExt()
return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);

Library pseudocode for shared/functions/extension/HaveRASExt

// HaveRASExt()
// ============

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) || boolean IMPLEMENTATION_DEFINED "Has RAS extension");

Library pseudocode for shared/functions/extension/HaveRNG

// HaveRNG()
// =========

// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.

boolean HaveRNG()
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension");

Library pseudocode for shared/functions/extension/HaveSBExt

// HaveSBExt()
// ===========

// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension");

Library pseudocode for shared/functions/extension/HaveSSBSExt

// HaveSSBSExt()
// =============

// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension");
Library pseudocode for shared/functions/extension/HaveSecureEL2Ext

// HaveSecureEL2Ext()
// ==================
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
{
    return HasArchVersion(ARMv8p4);
}

Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

// HaveSecureExtDebugView()
// ========================
// Returns TRUE if support for Secure and Non-secure views of debug peripherals is implemented.

boolean HaveSecureExtDebugView()
{
    return HasArchVersion(ARMv8p4);
}

Library pseudocode for shared/functions/extension/HaveSelfHostedTrace

// HaveSelfHostedTrace()
// =====================

boolean HaveSelfHostedTrace()
{
    return HasArchVersion(ARMv8p4);
}

Library pseudocode for shared/functions/extension/HaveSmallTranslationTblExt

// HaveSmallTranslationTblExt()
// ============================
// Returns TRUE if Small Translation Table Support is implemented.

boolean HaveSmallTranslationTableExt()
{
    return HasArchVersion(ARMv8p4) && boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension";
}

Library pseudocode for shared/functions/extension/HaveStage2MemAttrControl

// HaveStage2MemAttrControl()
// ==========================
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is implemented.

boolean HaveStage2MemAttrControl()
{
    return HasArchVersion(ARMv8p4);
}

Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

// HaveStatisticalProfiling()
// ==========================
// Returns TRUE if Statistical Profiling Extension is implemented,
// and FALSE otherwise.

boolean HaveStatisticalProfiling()
{
    return HasArchVersion(ARMv8p2);
}

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p1

// HaveStatisticalProfilingv1p1()
// ==============================
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p1()
{
    return (HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");
}
// HaveStatisticalProfilingv1p2()
// ==============================
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p2()
{
    return HasArchVersion(ARMv8p7) && HaveStatisticalProfiling() &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension";}

// HaveTME()
// =========

boolean HaveTME()
{
    return boolean IMPLEMENTATION_DEFINED "Has Transactional Memory extension";
}

// HaveTWEDExt()
// =============
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented, and FALSE otherwise.

boolean HaveTWEDExt()
{
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";
}

// HaveTraceBufferExtension()
// ==========================
// Returns TRUE if support for the Trace Buffer Extension is implemented. This feature depends upon
// the Secure External Debug view feature being implemented. Returns FALSE otherwise.

boolean HaveTraceBufferExtension()
{
    return HaveSecureExtDebugView() && boolean IMPLEMENTATION_DEFINED "Trace Buffer Extension implemented";
}

// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
{
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
}

// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
{
    return HasArchVersion(ARMv8p2);
}

// HaveUAOExt()
// ============

boolean HaveUAOExt()
{
    return HasArchVersion(ARMv8p2);
// HaveV82Debug()
// ==============
boolean HaveV82Debug()
return HasArchVersion(ARMv8p2);

// HaveVirtHostExt()
// =================
boolean HaveVirtHostExt()
return HasArchVersion(ARMv8p1);

// Havev85PMU()
// ============
// Returns TRUE if v8.5-Performance Monitor Unit extension support is implemented, and FALSE otherwise.
boolean Havev85PMU()
return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has PMUv3p5 extension";

// Havev8p4Debug()
// ===============
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.
boolean Havev8p4Debug()
return HasArchVersion(ARMv8p4);

// InsertIESBBeforeException
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable SError interrupt must be taken before executing any instructions in the exception handler. However, this can be before the branch to the exception handler is made.
boolean InsertIESBBeforeException(bits(2) el);
// BFAdd()
// =======
// Single-precision add following BFloat16 computation behaviors.

bits(32) BFAdd(bits(32) op1, bits(32) op2)
    result;
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN();
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0');    // Positive sign when Round to Odd
            else
                result = BFRound(result_value);
        end
    end
    return result;

Library pseudocode for shared/functions/float/bfloat/BFAdd

Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

// BFMatMulAdd()
// =============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)
    assert N == 128;
    bits(N) result;
    bits(32) sum, prod0, prod1;
    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                prod0 = BFMul(Elem[op1, 4*i + 2*k + 0, 16], Elem[op2, 4*j + 2*k + 0, 16]);
                prod1 = BFMul(Elem[op1, 4*i + 2*k + 1, 16], Elem[op2, 4*j + 2*k + 1, 16]);
                sum = BFAdd(sum, BFAdd(prod0, prod1));
                Elem[result, 2*i + j, 32] = sum;
        end
    end
    return result;

Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

Shared Pseudocode Functions
Library pseudocode for shared/functions/float/bfloat/BFMul

// BFMul()
// =======
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFMul(bits(16) op1, bits(16) op2)

    bits(32) result;
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN();
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = BFRound(value1*value2);
    end

    return result;

Library pseudocode for shared/functions/float/bfloat/BFMulAdd

// BFMulAdd()
// =======
// Used by BFMLALB and BFMLALT instructions.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)

    boolean altfp = HaveAltFP() && fpcr.AH == '1';  // When TRUE:
    boolean fpxc  = !altfp;                         // Do not generate floating point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then fpcr.RMode    = '00';             // Use RNE rounding mode
    return FPMulAdd(addend, op1, op2, fpcr, fpxc);
Library pseudocode for shared/functions/float/bfloat/BFRound

// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)
    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126;  E = 8;  F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign);  // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

    return result;
Library pseudocode for shared/functions/float/bfloat/BFUnpack

// BFUnpack()
// =========
// Unpacks a BFloat16 or single-precision value into its type, sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities, is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)
    assert N IN {16,32};
    if N == 16 then
        sign   = fpval<15>;
        exp    = fpval<14:7>;
        frac   = fpval<6:0> : Zeros(16);
    else // N == 32
        sign   = fpval<31>;
        exp    = fpval<30:23>;
        frac   = fpval<22:0>;
    if IsZero(exp) then
        fptype = FPType_Zero;  value = 0.0;    // Fixed Flush to Zero
    elsif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType_Infinity;  value = 2.0^1000000; // no SNaN for BF16 arithmetic
        else
            fptype = FPType_QNaN; value = 0.0;
        end
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);
    end
    if sign == '1' then value = -value;
    return (fptype, sign, value);
Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```c
// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value with using rounding mode of
// Round to Nearest Even when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr, FPRounding rounding)
    bits(32) result;                                // BF16 value in top 16 bits
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then rounding = FPRounding_TIEEVEN;    // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        end if
    end if
    if fptype == FPTYPE_SNaN then
        if fpexc then
            FPProcessException(FPExc_InvalidOp, fpcr);
        end if
    end if
    elsif fptype == FPTYPE_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPTYPE_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCVBF(value, fpcr, rounding, fpexc);
    end if

    // Returns correctly rounded BF16 value from top 16 bits
    return result<31:16>;

// FPConvertBF()
// =============
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for shared/functions/float/bfloat/FPRoundCVBF

```c
// FPRoundCVBF()
// =============
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(32) FPRoundCVBF(real op, FPCRType fpcr, FPRounding rounding, boolean fpexc)
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);
```
Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

// FixedToFP()
// ===========
// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
assert N IN {16,32,64};
assert M IN {16,32,64};
bits(N) result;
assert fbits >= 0;
assert rounding != FPRounding_ODD;
// Correct signed-ness
int_operand = Int(op, unsigned);
// Scale by fractional bits and generate a real value
real_operand = Real(int_operand) / 2.0^fbits;
if real_operand == 0.0 then
  result = FPZero('0');
else
  result = FPRound(real_operand, fpcr, rounding);
return result;

Library pseudocode for shared/functions/float/fpabs/FPAbs

// FPAbs()
// ========

bits(N) FPAbs(bits(N) op)
assert N IN {16,32,64};
if !UsingAArch32() & HaveAltFP() then
  FPCRType fpcr = FPCR[];
  if fpcr.AH == '1' then
    (ftype, _, _) = FPUnpack(op, fpcr, FALSE);
    if ftype IN {FPType_SNaN, FPType_QNaN} then
      return op; // When fpcr.AH=1, sign of NaN has no consequence
  end if
return '0' : op<N-2:0>;
Library pseudocode for shared/functions/float/fpadd/FPAdd

// FPAdd()
// ========

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1  = (type1 == FPType_Infinity);  inf2  = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);      zero2 = (type2 == FPType_Zero);
  if inf1 && inf2 && sign1 == NOT(sign2) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
    result = FPInfinity('0');
  elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
    result = FPInfinity('1');
  elsif zero1 && zero2 && sign1 == sign2 then
    result = FPZero(sign1);
  else
    result_value = value1 + value2;
    if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr, rounding);
    end
    FPProcessDenorms(type1, type2, N, fpcr);
  end
return result;

Library pseudocode for shared/functions/float/fpcompare/FPCompare

// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
  result = '0011';
  if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
    FPProcessException(FPExc_InvalidOp, fpcr);
  end
else
  // All non-NaN cases can be evaluated on the values produced by FPUnpack()
  if value1 == value2 then
    result = '0110';
  elsif value1 < value2 then
    result = '1000';
  else  // value1 > value2
    result = '0010';
    FPProcessDenorms(type1, type2, N, fpcr);
end
return result;
Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

```plaintext
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
result = FALSE;
if type1 == FPType_SNaN || type2 == FPType_SNaN then
FPProcessException(FPExc_InvalidOp, fpcr);
else
  // All non-NaN cases can be evaluated on the values produced by FPUnpack()
  result = (value1 == value2);
FPProcessDenorms(type1, type2, N, fpcr);

return result;
```

Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```plaintext
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
result = FALSE;
FPProcessException(FPExc_InvalidOp, fpcr);
else
  // All non-NaN cases can be evaluated on the values produced by FPUnpack()
  result = (value1 >= value2);
FPProcessDenorms(type1, type2, N, fpcr);

return result;
```

Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```plaintext
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);

if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
result = FALSE;
FPProcessException(FPExc_InvalidOp, fpcr);
else
  // All non-NaN cases can be evaluated on the values produced by FPUnpack()
  result = (value1 > value2);
FPProcessDenorms(type1, type2, N, fpcr);

return result;
```
Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (ftype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        end
    elsif fptype == FPTypeInfinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
        end
    else
        result = FPRoundCV(value, fpcr, rounding);
        FPProcessDenorm(ftype, N, fpcr);
    end

    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr)

    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```
Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)
assert N IN {16,32,64};
assert M IN {16,32,64};
bits(M) result;
bits(51) frac;

sign = op<N-1>;

// Unpack payload from input NaN
case N of
  when 64 frac = op<50:0>;
  when 32 frac = op<21:0>:Zeros(29);
  when 16 frac = op<8:0>:Zeros(42);

// Repack payload into output NaN, while
// converting an SNaN to a QNaN.
case M of
  when 64 result = sign:Ones(M-52):frac;
  when 32 result = sign:Ones(M-23):frac<50:29>;
  when 16 result = sign:Ones(M-10):frac<50:42>;
return result;

Library pseudocode for shared/functions/float/fpcrtype/FPCRType

type FPCRType;

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

// FPDecodeRM()
// ============
// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
case rm of
  when '00' result = FPRounding_TIEAWAY; // A
  when '01' result = FPRounding_TIEEVEN; // N
  when '10' result = FPRounding_POSINF; // P
  when '11' result = FPRounding_NEGINF; // M
return result;

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

// FPDecodeRounding()
// ===============
// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
case rmode of
  when '00' return FPRounding_TIEEVEN; // N
  when '01' return FPRounding_POSINF; // P
  when '10' return FPRounding_NEGINF; // M
  when '11' return FPRounding_ZERO; // Z
Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```plaintext
// FPDefaultNaN()
// =============

bits(N) FPDefaultNaN()
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
bit sign;
if !UsingAArch32() && HaveAltFP() then
    FPCRType fpcr = FPCR[];
sign = if fpcr.AH == '1' then '1' else '0';
else
    sign = '0';
bits(E) exp = Ones(E);
bits(F) frac = '1':Zeros(F-1);
return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpdiv/FPDiv

```plaintext
// FPDiv()
// ======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
    inf1  = type1 == FPType_Infinity;
    inf2  = type2 == FPType_Infinity;
    zero1 = type1 == FPType_Zero;
    zero2 = type2 == FPType_Zero;
    if (inf1 && inf2) || (zero1 && zero2) then
        result = FPDefaultNaN();
    elsif inf1 || zero2 then
        result = FPInfinity(sign1 EOR sign2);
        if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif zero1 || inf2 then
        result = FPZero(sign1 EOR sign2);
    else
        result = FPRound(value1/value2, fpcr);
    if !zero2 then
        FPProcessDenorms(type1, type2, N, fpcr);
return result;
```

Library pseudocode for shared/functions/float/fpexc/FPExc

```plaintext
enumeration FPExc       {
    FPExc_InvalidOp,
    FPExc_DivideByZero,
    FPExc_Overflow,
    FPExc_Underflow,
    FPExc_Inexact,
    FPExc_InputDenorm;
```
Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```plaintext
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
bits(E) exp = Ones(E);
bits(F) frac = Zeros(F);
return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```plaintext
// FPMatMulAdd()
// =============
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRType fpcr)
assert N == esize * 2 * 2;
bits(N) result;
bits(esize) prod0, prod1, sum;
for i = 0 to 1
    for j = 0 to 1
        sum = Elem[addend, 2*i + j, esize];
        prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                       Elem[op2, 2*j + 0, esize], fpcr);
        prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                       Elem[op2, 2*j + 1, esize], fpcr);
        sum = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
        Elem[result, 2*i + j, esize] = sum;
return result;
```
Library pseudocode for shared/functions/float/fpmax/FPMax

// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
  boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
  return FPMax(op1, op2, fpcr, altfp);

// FPMax()
// =======

// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behaviour.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean altfp)
  assert N IN {16,32,64};
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);
  if (altfp && type1 == FPType_Zero && type2 == FPType_Zero &&
      ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
    return FPZero(sign2);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);
  if !done then
    if value1 > value2 then
      (fptype,sign,value) = (type1,sign1,value1);
    else
      (fptype,sign,value) = (type2,sign2,value2);
    if fptype == FPType_Infinity then
      result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
      sign = sign1 AND sign2;         // Use most positive sign
      result = FPZero(sign);
    else
      // The use of FPRound() covers the case where there is a trapped underflow exception
      // for a denormalized number even though the result is exact.
      rounding = FPRoundingMode(fpcr);
      if altfp then // Denormal output is not flushed to zero
        fpcr.FZ = '0';
        fpcr.FZ16 = '0';
      result = FPRound(value, fpcr, rounding, TRUE);
    FPProcessDenorms(type1, type2, N, fpcr);
  return result;

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign)
  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = N - (E + 1);
  exp = Ones(E-1):'0';
  frac = Ones(F);
  return sign : exp : frac;
Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```plaintext
// FPMaxNum()
// =========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

if !(altfp && type1_nan && type2_nan) then
  // Treat a single quiet-NaN as -Infinity.
  if type1 == FPType_QNaN && type2 != FPType_QNaN then
    op1 = FPInfinity('1');
  elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
    op2 = FPInfinity('1');
  else
    altfmaxfmin = FALSE;  // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin);
  fi
fi
return result;
```

Library pseudocode for shared/functions/float/fpmerge/IsMerging

```plaintext
// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRType fpcr)
  boolean merge = HaveAltFP() && !UsingAArch32() && fpcr.NEP == '1';
  return merge;
```
Library pseudocode for shared/functions/float/fpmin/FPMin

// FPMin()
// ========
bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTyp e fpcr)
boolean altfp = HaveAltFP() & & !UsingAArch32() & & fpcr.AH == '1';
return FPMin(op1, op2, fpcr, altfp);

// FPMin()
// ========
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behaviour.
bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTyp e fpcr, boolean altfp)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
if (altfp & & type1 == FPTyp e_Zero & & type2 == FPTyp e_Zero & &
    ((sign1 == '0' & & sign2 == '1') || (sign1 == '1' & & sign2 == '0'))) then
    return FPZero(sign2);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);
if !done then
if value1 < value2 then
    (ftype,sign,value) = (type1,sign1,value1);
else
    (ftype,sign,value) = (type2,sign2,value2);
if ftype == FPTyp e_Infinity then
    result = FPInfinity(sign);
elsif ftype == FPTyp e_Zero then
    sign = sign1 OR sign2;              // Use most negative sign
    result = FPZero(sign);
else
    // The use of FPRound() covers the case where there is a trapped underflow exception
    // for a denormalized number even though the result is exact.
    rounding = FPRoundingMode(fpcr);
    if altfp then                      // Denormal output is not flushed to zero
        fpcr.FZ = '0';
        fpcr.FZ16 = '0';
    result = FPRound(value, fpcr, rounding, TRUE);
FPProcessDenorms(type1, type2, N, fpcr);
return result;
// FPMinNum()
// =========

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,-,-) = FPUnpack(op1, fpcr);
(type2,-,-) = FPUnpack(op2, fpcr);

boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

if !(altfp && type1_nan && type2_nan) then
  // Treat a single quiet-NaN as +Infinity.
  if type1 == FPType_QNaN && type2 != FPType_QNaN then
    op1 = FPInfinity('0');
  elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
    op2 = FPInfinity('0');

  altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
  result = FPMin(op1, op2, fpcr, altfmaxfmin);

  return result;

Library pseudocode for shared/functions/float/fpmul/FPMul

// FPMul()
// ========

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);

  if (inf1 && zero2) || (zero1 && inf2) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif inf1 || inf2 then
    result = FPInfinity(sign1 EOR sign2);
  elsif zero1 || zero2 then
    result = FPZero(sign1 EOR sign2);
  else
    result = FPRound(value1*value2, fpcr);

  FPProcessDenoms(type1, type2, N, fpcr);

  return result;
Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd
// FPMulAdd()
// =========

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
boolean fpexc = TRUE; // Generate floating-point exceptions
return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// =========
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)
assert N IN {16,32,64};

(typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
(type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
(type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
rounding = FPRoundingMode(fpcr);
inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

(done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);
if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
  if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  if done then
    infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
    // Determine sign and type product will have if it does not cause an
    // Invalid Operation.
    signP = sign1 EOR sign2;
    infP = inf1 || inf2;
    zeroP = zero1 || zero2;
    // Non SNaN-generated Invalid Operation cases are multiplies of zero
    // by infinity and additions of opposite-signed infinities.
    invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);
    if invalidop then
      result = FPDefaultNaN();
      if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
      // Other cases involving infinities produce an infinity of the same sign.
      elseif (infA && signA == '0') || (infP && signP == '0') then
        result = FPInfinity('0');
      elseif (infA && signA == '1') || (infP && signP == '1') then
        result = FPInfinity('1');
      // Cases where the result is exactly zero and its sign is not determined by the
      // rounding mode are additions of same-signed zeros.
      elseif zeroA && zeroP && signA == signP then
        result = FPZero(signA);
      // Otherwise calculate numerical result and round it.
      else
        result_value = valueA + (value1 * value2);
        if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
          result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
        result = FPZero(result_sign);
      else
        result = FPRound(result_value, fpcr, rounding, fpexc);
  if !invalidop && fpexc then
    FPProcessDenorms3(typeA, type1, type2, N, fpcr);
Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
assert N == 32;
rounding = FPRoundingMode(fpcr);
(typeA,signA,valueA) = FPUnpack(addend, fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
(done, result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);

if !((HaveAltFP() && !UsingAArch32() && fpcr.AH == '1')) then
    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elseif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elseif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elseif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);

        if !invalidop then
            FPProcessDenorm(typeA, N, fpcr);

        return result;
    end if
end if
// FPProcessNaNs3H()
// =================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
    bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
    FPCRType fpcr)

    assert N IN {32,64};

    bits(N) result;
    // When TRUE, use alternative NaN propagation rules.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};
    boolean fpexc = TRUE;
    if altfp then
        if (type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN) then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;
        if altfp && op1_nan && op2_nan && op3_nan then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc)); // <n> register
        elsif altfp && op2_nan && (op1_nan || op3_nan) then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc)); // <n> register
        elsif altfp && op3_nan && op1_nan then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpcr, fpexc)); // <m> register
        elsif type1 == FPType_SNaN then
            done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
        elsif type2 == FPType_SNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc));
        elsif type3 == FPType_SNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc));
        elsif type1 == FPType_QNaN then
            done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
        elsif type2 == FPType_QNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc));
        elsif type3 == FPType_QNaN then
            done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc));
        else
            done = FALSE; result = Zeros(); // 'Don't care' result
    return (done, result);
Library pseudocode for shared/functions/float/fpmulx/FPMulX

// FPMulX()  
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPRound(value1*value2, fpcr);
        else
            FPProcessDenorms(type1, type2, N, fpcr);
    return result;

Library pseudocode for shared/functions/float/fpneg/FPNeg

// FPNeg()  
// =======

bits(N) FPNeg(bits(N) op)
assert N IN {16,32,64};
if !UsingAArch32() && HaveAltFP() then
    FPCRType fpcr = FPCR[];
    if fpcr.AH == '1' then
        (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
        if fptype IN {FPType_SNaN, FPType_QNaN} then
            return op;        // When fpcr.AH=1, sign of NaN has no consequence
        return NOT(op<N-1>) : op<N-2:0>;

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

// FPOnePointFive()
// ===============

bits(N) FPOnePointFive(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '0':Ones(E-1);
frac = '1':Zeros(F-1);
result = sign : exp : frac;
return result;
Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorm

```plaintext
// FPProcessDenorm()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPType ftype, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms

```plaintext
// FPProcessDenorms()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms(FPType type1, FPType type2, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms3

```plaintext
// FPProcessDenorms3()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms3(FPType type1, FPType type2, FPType type3, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
                           type3 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```plaintext
// FPProcessException()
// ====================
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp        cumul = 0;
        when FPExc_DivideByZero     cumul = 1;
        when FPExc_Overflow         cumul = 2;
        when FPExc_Underflow        cumul = 3;
        when FPExc_Inexact          cumul = 4;
        when FPExc_InputDenorm      cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrappedException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    return;
```
Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr) {
    boolean fpexc = TRUE;  // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);
}

// FPProcessNaN()
// ==============
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr, boolean fpexc) {
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
}
Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

// FPProcessNaNs()
// ===============

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2, FPCRType fpcr)

  boolean altfmaxfmin = FALSE;    // Do not use alfp mode for FMIN, FMAX and variants
  boolean fpexc = TRUE;     // Generate floating-point exceptions
  return FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpexc);

// FPProcessNaNs()
// ===============

// The boolean part of the return value says whether a NaN has been found and processed. The bits(N) part is only relevant if it has and supplies the result of the operation.

// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls alternative floating-point behaviour for FMAX, FMIN and variants. 'fpexc' controls the generation of floating-point exceptions. Status information is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2, FPCRType fpcr, boolean altfmaxfmin, boolean fpexc)

  assert N IN {16,32,64};
  boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
  boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
  boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
  boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;
  FPType type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

  if altfmaxfmin && (op1_nan || op2_nan) then
    FPProcessException(FPExc_InvalidOp, fpcr);
  elseif altfp && op1_nan && op2_nan then
    done = TRUE; result = FPProcessNaN(type_nan, op1, fpcr, fpexc); // <n> register NaN selected
  elseif type1 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
  elseif type2 == FPType_SNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
  elseif type1 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
  elseif type2 == FPType_QNaN then
    done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
  else
    done = FALSE; result = Zeros(); // 'Don't care' result

  return (done, result);
// FPProcessNaNs3()
// ================
(boolean, bits(N)) FPProcessNaNs3(FPTypetype1, FPTypetype2, FPTypetype3,
Bits(N)op1, Bits(N)op2, Bits(N)op3,
FPCRTypefpcr)
    booleanfpexc = TRUE;  // Generate floating-point exceptions
    return FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);
// FPProcessNaNs3()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.
(boolean, bits(N)) FPProcessNaNs3(FPTypetype1, FPTypetype2, FPTypetype3,
    Bits(N)op1, Bits(N)op2, Bits(N)op3,
    FPCRTypefpcr, booleanfpexc)
    assert N IN {16,32,64};
    booleanop1nan = type1 IN {FPTypetype_SNaN, FPTypetype_QNaN};
    booleanop2nan = type2 IN {FPTypetype_SNaN, FPTypetype_QNaN};
    booleanop3nan = type3 IN {FPTypetype_SNaN, FPTypetype_QNaN};
    booleanaltfp = HaveAltFP() && !UsingAAArch32() && fpcr.AH == '1';
    if altfp then
        if type1 == FPTypetype_SNaN || type2 == FPTypetype_SNaN || type3 == FPTypetype_SNaN then
            type_nan = FPTypetype_SNaN;
        else
            type_nan = FPTypetype_QNaN;
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
// FPRecipEstimate()
// ================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
assert N IN {16,32,64};

// When using alternative floating-point behaviour, do not generate
// floating-point exceptions, flush denormal input and output to zero,
// and use RNE rounding mode.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpexc = '!altfp;
if altfp then fpcr.<FIZ,FZ> = '11';
if altfp then fpcr.RMode    = '00';

(fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);
FPRounding rounding = FPRoundingMode(fpcr);
if fptype == FPTag_SNaN || fptype == FPTag_QNaN then
  result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elsif fptype == FPTag_Infinity then
  result = FPZero(sign);
elsif fptype == FPTag_Zero then
  result = FPInfinity(sign);
if fpexc then
  FPProcessException(FPEX_DivideByZero, fpcr);
elsif ((N == 16 && Abs(value) < 2.0^-16) ||
  (N == 32 && Abs(value) < 2.0^-128) ||
  (N == 64 && Abs(value) < 2.0^-1024)) then
  case rounding of
    when FPRounding_TIEEVEN
      overflow_to_inf = TRUE;
    when FPRounding_POSINF
      overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
      overflow_to_inf = (sign == '1');
    when FPRounding_ZERO
      overflow_to_inf = FALSE;
  result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
  if fpexc then
    FPProcessException(FPEX_Overflow, fpcr);
    FPProcessException(FPEX_Inexact, fpcr);
  elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
    && ((N == 16 && Abs(value) >= 2.0^14) ||
    (N == 32 && Abs(value) >= 2.0^126) ||
    (N == 64 && Abs(value) >= 2.0^1022))
  then
    // Result flushed to zero of correct sign
    result = FPZero(sign);
    // Flush-to-zero never generates a trapped exception.
    if UsingAArch32() then
      FPSR.UFC = '1';
    else
      if fpexc then FPSR.UFC = '1';
else
  // Scale to a fixed point value in the range 0.5 ≤ x < 1.0 in steps of 1/512, and
  // calculate result exponent. Scaled value has copied sign bit,
  // exponent = 1022 = double-precision biased version of -1,
  // fraction = original fraction
  case N of
    when 16
      fraction = operand<9:0> : Zeros(42);
      exp = UInt(operand<14:10>);
    when 32
      fraction = operand<22:0> : Zeros(29);
      exp = UInt(operand<30:23>);
    when 64
      fraction = operand<51:0>;
  Shared Pseudocode Functions Page 3352
exp = UInt(operand<62:52>);

if exp == 0 then
    if fraction<51> == '0' then
        exp = -1;
        fraction = fraction<49:0>:'00';
    else
        fraction = fraction<50:0>:'0';

integer scaled;
boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

if !increasedprecision then
    scaled = UInt('1':fraction<51:44>);
else
    scaled = UInt('1':fraction<51:41>);

case N of
    when 16 result_exp =  29 - exp; // In range 29-30 = -1 to 29+1 = 30
    when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
    when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

    // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
    // fixed-point number in range [0.5 .. 1.0].
    estimate = RecipEstimate(scaled, increasedprecision);

    // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
    // fixed-point result in the range [1.0 .. 2.0].
    // Convert to scaled floating point result with copied sign bit,
    // high-order bits from estimate, and exponent calculated above.
    if !increasedprecision then
        fraction = estimate<7:0> : Zeros(44);
    else
        fraction = estimate<11:0> : Zeros(40);

    if result_exp == 0 then
        fraction = '1' : fraction<51:1>;
    elsif result_exp == -1 then
        fraction = '01' : fraction<51:2>;
        result_exp = 0;

    case N of
        when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
        when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
        when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

return result;
RecipEstimate()
// Compute estimate of reciprocal of 9-bit fixed-point number.
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a, boolean increasedprecision)
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;   // Round to nearest
        a = a*2+1;
        integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2;            // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;  // Round to nearest
        a = a*2+1;
        real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        real error = real_val - Real(r);
        boolean round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
        if round_up then r = r+1;
        assert 4096 <= r && r < 8192;
    return r;
Library pseudocode for shared/functions/fp/fprecpx/FPRecpX

// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;
    bits(N)           result;
    bits(esize)       exp;
    bits(esize)       max_exp;
    bits(N-(esize+1)) frac = Zeros();
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;
    max_exp = Ones(esize) - 1;
    if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;
    return result;
// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding, boolean fpexc)
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
// FPRoundBase()
// =============

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat16)
  boolean fpexc = TRUE; // Generate floating-point exceptions
  return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// FPRoundBase()
// =============

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat16, boolean fpexc)
  assert N IN {16,32,64};
  assert op != 0.0;
  assert rounding != FPRounding_TIEAWAY;
  bits(N) result;

  // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
  if N == 16 then
    minimum_exp = -14;  E = 5;  F = 10;
  elsif N == 32 && isbfloat16 then
    minimum_exp = -126;  E = 8;  F = 7;
  elsif N == 32 then
    minimum_exp = -126;  E = 8;  F = 23;
  else  // N == 64
    minimum_exp = -1022;  E = 11;  F = 52;

  // Split value into sign, unrounded mantissa and exponent.
  if op < 0.0 then
    sign = '1';  mantissa = -op;
  else
    sign = '0';  mantissa = op;
  exponent = 0;
  while mantissa < 1.0 do
    mantissa = mantissa * 2.0;  exponent = exponent - 1;
  while mantissa >= 2.0 do
    mantissa = mantissa / 2.0;  exponent = exponent + 1;
  // When TRUE, detection of underflow occurs after rounding and the test for a
  // denormalized number for single and double precision values occurs after rounding.
  altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

  // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
  if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
    exponent < minimum_exp) then
    // Flush-to-zero never generates a trapped exception.
    if UsingAArch32() then
      FPSCR.UFC = '1';
    else
      FPSR.UFC = '1';
    return FPZero(sign);
  biased_exp_unconstrained = exponent - minimum_exp + 1;
  int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
  error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

  // Start creating the exponent value for the result. Start by biasing the actual exponent
  // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
  biased_exp = Max(exponent - minimum_exp + 1, 0);
  if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

  // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
  int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
  error = mantissa * 2.0^F - Real(int_mant);
// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
if !altfp && biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
  if fpexc then FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
if altfp then
  case rounding of
    when FPRounding_TIEEVEN
      round_up_unconstrained = (error_unconstrained > 0.5 ||
                               (error_unconstrained == 0.5 && int_mant_unconstrained < 0) == '1'));
      round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
      overflow_to_inf = TRUE;
    when FPRounding_POSINF
      round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
      round_up = (error != 0.0 && sign == '0');
      overflow_to_inf = (sign == '0');
    when FPRounding_NEGINF
      round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
      round_up = (error != 0.0 && sign == '1');
      overflow_to_inf = (sign == '1');
    when FPRounding_ZERO, FPRounding_ODD
      round_up_unconstrained = FALSE;
      round_up = FALSE;
      overflow_to_inf = FALSE;
  if round_up_unconstrained then
    int_mant_unconstrained = int_mant_unconstrained + 1;
    if int_mant_unconstrained == 2^(F+1) then // Rounded up to next exponent
      biased_exp_unconstrained = biased_exp_unconstrained + 1;
      int_mant_unconstrained = int_mant_unconstrained DIV 2;

  // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
  if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
    // the result of unconstrained rounding is less than the minimum normalized number
    if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then // Flush-to-zero
      if fpexc then
        FPSR.UFC = '1';
        FPProcessException(FPExc_Inexact, fpcr);
        return FPZero(sign);
      elsif error != 0.0 || fpcr.UFE == '1' then
        if fpexc then FPProcessException(FPExc_Underflow, fpcr);
  else // altfp == FALSE
    case rounding of
      when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
        overflow_to_inf = TRUE;
      when FPRounding_POSINF
        round_up = (error != 0.0 && sign == '0');
        overflow_to_inf = (sign == '0');
      when FPRounding_NEGINF
        round_up = (error != 0.0 && sign == '1');
        overflow_to_inf = (sign == '1');
      when FPRounding_ZERO, FPRounding_ODD
        round_up = FALSE;
        overflow_to_inf = FALSE;
    if round_up then
      int_mant = int_mant + 1;
      if int_mant == 2^F then // Rounded up from denormalized to normalized
        biased_exp = 1;
      if int_mant == 2^(F+1) then // Rounded up to next exponent
        biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;

    // Handle rounding to odd
    if error != 0.0 && rounding == FPRounding_ODD then
      int_mant<0> = '1';
// Deal with overflow and generate result.
if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
  if biased_exp >= 2^E - 1 then
    result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
    error = 1.0;  // Ensure that an Inexact exception occurs
  else
    result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
else  // Alternative half precision
  if biased_exp >= 2^E then
    result = sign : Ones(N-1);
    if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    error = 0.0;  // Ensure that an Inexact exception does not occur
  else
    result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

// Deal with Inexact exception.
if error != 0.0 then
  if fpexc then FPProcessException(FPExc_Inexact, fpcr);
return result;

Library pseudocode for shared/functions/float/fpround/FPRoundCV

// FPRoundCV()
//============
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr, FPRounding rounding)
  fpcr.FZ16 = '0';
  boolean fpexc = TRUE;  // Generate floating-point exceptions
  boolean isbfloat16 = FALSE;
  return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

Library pseudocode for shared/functions/float/fprounding/FPRounding

enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
  FPRounding_NEGINF, FPRounding_ZERO,
  FPRounding_TIEAWAY, FPRounding_ODD};

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

// FPRoundingMode()
//============
// Return the current floating-point rounding mode.
FPRounding FPRoundingMode(FPCRType fpcr)
  return FPDecodeRounding(fpcr.RMode);
// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRTy pe fpcr, FPRounding rounding, boolean exact)
assert rounding != FPRounding_ODD;
assert N IN {16,32,64};

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAAArch32() && fpcr.AH == '1';
fpexc = !altfp;

// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

if fptype == FPTye_SNaN || fptype == FPTye_ONaN then
    result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPTye_Infinity then
    result = FPInfinity(sign);
elsif fptype == FPTye_Zero then
    result = FPZero(sign);
else
    // Extract integer component.
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
    when FPRounding_POSINF
        round_up = (error != 0.0);
    when FPRounding_NEGINF
        round_up = FALSE;
    when FPRounding_ZERO
        round_up = (error != 0.0 && int_result < 0);
    when FPRounding_TIEAWAY
        round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
    if round_up then int_result = int_result + 1;

    // Convert integer value into an equivalent real value.
    real_result = Real(int_result);

    // Re-encode as a floating-point value, result is always exact.
    if real_result == 0.0 then
        result = FPZero(sign);
    else
        result = FPRound(real_result, fpcr, FPRounding_ZERO);

    // Generate inexact exceptions.
    if error != 0.0 && exact then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
Library pseudocode for shared/functions/float/froundintn/FPRoundIntN
// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
assert rounding != FPRounding_ODD;
assert N IN {32,64};
assert intsize IN {32, 64};
integer exp;
constant integer E = (if N == 32 then 8 else 11);
constant integer F = N - (E + 1);

// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;

// Unpack using FPCR to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
  if N == 32 then
    exp = 126 + intsize;
    result = '1':exp<(E-1):0>:Zeros(F);
  else
    exp = 1022+intsize;
    result = '1':exp<(E-1):0>:Zeros(F);
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif fptype == FPType_Zero then
    result = FPZero(sign);
  else
    // Extract integer component.
    int_result = RoundDown(value);
    error = value - Real(int_result);
    // Determine whether supplied rounding mode requires an increment.
    case rounding of
      when FPRounding_TIEEVEN
        round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
      when FPRounding_POSINF
        round_up = error != 0.0;
      when FPRounding_NEGINF
        round_up = FALSE;
      when FPRounding_TIEAWAY
        round_up = error > 0.5 || (error == 0.5 && int_result >= 0);
    if round_up then int_result = int_result + 1;
    overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);
    if overflow then
      if N == 32 then
        exp = 126 + intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
      else
        exp = 1022 + intsize;
        result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
      // This case shouldn't set Inexact.
      error = 0.0;
    else
      // Convert integer value into an equivalent real value.
      real_result = Real(int_result);
      // Re-encode as a floating-point value, result is always exact.
      if real_result == 0.0 then
        result = FPZero(sign);
      else
        result = FPRound(real_result, fpcr, FPRounding_ZERO);
  // Shared Pseudocode Functions
  Page 3363
// Generate inexact exceptions.
if error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);

return result;
// FPRSQRTEstimate()
// ===============

bits(N) FPRSQRTEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};

    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions and flush denormal input to zero.
    boolean altpf = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altpf;
    if altpf then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    if fptype == FPTypemSNaN || fptype == FPTypemQNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPTypemZero then
        result = FPIInfinity(sign);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPTypemInfinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);
            if exp == 0 then
                while fraction<51> == '0' do
                    fraction = fraction<50:0> : '0';
                    exp = exp - 1;
                    fraction = fraction<50:0> : '0';

        integer scaled;
        boolean increasedprecision = N==32 && HaveFeatRPRES() && altpf;

        if !increasedprecision then
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:44>);
            else
                scaled = UInt('01':fraction<51:45>);
            else
                if exp<0> == '0' then
                    scaled = UInt('1':fraction<51:41>);
                else
                    scaled = UInt('01':fraction<51:42>);
        case N of
            when 16 result_exp = ( 44 - exp) DIV 2;
            when 32 result_exp = (380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;
        estimate = RecipSQRTEstimate(scaled, increasedprecision);

    // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
    // fixed-point result in the range [1.0 .. 2.0].
// Convert to scaled floating point result with copied sign bit and high-order
// fraction bits, and exponent calculated above.

case N of
    when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>: Zeros(2);
    when 32
        if !increasedprecision then
            result = '0' : result_exp<N-25:0> : estimate<7:0>: Zeros(15);
        else
            result = '0' : result_exp<N-25:0> : estimate<11:0>: Zeros(11);
    when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>: Zeros(44);

return result;
// RecipSqrtEstimate()
// ===================
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
// a is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a, boolean increasedprecision)
{
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then                      // 0.25 .. 0.5
            a = a*2+1;                       // a in units of 1/512 rounded to nearest
            integer b = 512;
            while a*(b+1)*(b+1) < 2^28 do
                b = b+1;
            // b = largest b such that b < 2^14 / sqrt(a)
            r = (b+1) DIV 2;                     // Round to nearest
            assert 256 <= r && r < 512;
        else                                 // 0.5 .. 1.0
            a = (a >> 1) << 1;               // Discard bottom bit
            a = (a+1)*2;                     // a in units of 1/256 rounded to nearest
            integer b = 512;
            while a*(b+1)*(b+1) < 2^28 do
                b = b+1;
            // b = largest b such that b < 2^14 / sqrt(a)
            r = (b+1) DIV 2;                     // Round to nearest
            assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;
        if a < 2048 then                     // 0.25... 0.5
            a = a*2 + 1;                     // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;          // This number will lie in the range of 32 to 64
        else                                 // 0.5..1.0
            a = (a >> 1) << 1;               // Discard bottom bit
            a = a+1;                         // Taking 10 bits of the fraction and force a 1 at the bottom
            real_val = Real(a);
            real_val = Sqrt(real_val);       // This number will lie in the range of 32 to 64
            real_val = real_val * Real(2^47); // The integer is the size of the whole DP mantissa
            int_val = RoundDown(real_val);   // Calculate rounding value
        error = real_val - Real(int_val);  // Error cannot be exactly 0.5 so do not need tie case
        round_up = error > 0.5;
        if round_up then int_val = int_val+1;
    real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
    int_val = RoundDown(real_val);       // Round that (to nearest even) to give integer
    error = real_val - Real(int_val);    // Error cannot be exactly 0.5 so do not need tie case
    round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
    if round_up then int_val = int_val+1;
    r = int_val;
    assert 4096 <= r && r < 8192;
    return r;
}
Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
assert N IN {16,32,64};
(fptype,sign,value) = FPUnpack(op, fpcr);

if fptype == FPType_SNaN || fptype == FPType_QNaN then
  result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPType_Zero then
  result = FPZero(sign);
elsif fptype == FPType_Infinity && sign == '0' then
  result = FPInfinity(sign);
elsif sign == '1' then
  result = FPDefaultNaN();
  FPProcessException(FPExc_InvalidOp, fpcr);
else
  result = FPRound(Sqrt(value), fpcr);
  FPProcessDenorm(fptype, N, fpcr);
return result;

Library pseudocode for shared/functions/float/fpsub/FPSub

// FPSub()
// ========

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
rounding = FPRoundingMode(fpcr);
(type1,sign1,value1) = FPUnpack(op1, fpcr);
(type2,sign2,value2) = FPUnpack(op2, fpcr);
(done, result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
if !done then
  inf1 = (type1 == FPType_Infinity);
  inf2 = (type2 == FPType_Infinity);
  zero1 = (type1 == FPType_Zero);
  zero2 = (type2 == FPType_Zero);

  if inf1 && inf2 && sign1 == sign2 then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
  elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
    result = FPInfinity('0');
  elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
    result = FPInfinity('1');
  elsif zero1 && zero2 && sign1 == NOT(sign2) then
    result = FPZero(sign1);
  else
    result_value = value1 - value2;
    if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
      result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
      result = FPZero(result_sign);
    else
      result = FPRound(result_value, fpcr, rounding);
      FPProcessDenorms(type1, type2, N, fpcr);
return result;
Library pseudocode for shared/functions/float/fpthree/FPThree

// FPThree()
// =========

bits(N) FPThree(bit sign)
assert N IN {16,32,64};
constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
constant integer F = N - (E + 1);
exp = '1':Zeros(E-1);
frac = '1':Zeros(F-1);
result = sign : exp : frac;
return result;

Library pseudocode for shared/functions/float/fptofixed/FPToFixed

// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and Rounding.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
assert N IN {16,32,64};
assert M IN {16,32,64};
assert fbits >= 0;
assert rounding != FPRounding_ODD;
// When alternative floating-point support is TRUE, do not generate
// Input Denormal floating-point exceptions.
altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
fpexc = !altfp;
// Unpack using fpcr to determine if subnormals are flushed-to-zero.
(fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
// If NaN, set cumulative flag or take exception.
if fptype == FPType_SNaN || fptype == FPType_QNaN then
    FPProcessException(FPExc_InvalidOp, fpcr);
// Scale by fractional bits and produce integer rounded towards minus-infinity.
value = value * 2.0^fbits;
int_result = RoundDown(value);
error = value - Real(int_result);
// Determine whether supplied rounding mode requires an increment.
case rounding of
    when FPRounding_TIEEVEN
        round_up = (error > 0.5 || (error == 0.5 & int_result<0> == '1'));
    when FPRounding_POSINF
        round_up = (error != 0.0);
    when FPRounding_NEGINF
        round_up = FALSE;
    when FPRounding_ZERO
        round_up = (error != 0.0 & int_result < 0);
    when FPRounding_TIEAWAY
        round_up = (error > 0.5 || (error == 0.5 & int_result >= 0));
if round_up then int_result = int_result + 1;
// Generate saturated result and exceptions.
(overflow) = SatQ(int_result, M, unsigned);
if overflow then
    FPProcessException(FPExc_InvalidOp, fpcr);
elself error != 0.0 then
    FPProcessException(FPExc_Inexact, fpcr);
return result;
 Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64)
    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc_idenorm);

    Z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        Z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        Z = '0';
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        Z = '0';
    if fptype == FPTypeInfinity then result = 0;
    return (result<N-1:0>, Z);

 Library pseudocode for shared/functions/float/fptwo/FPTwo

// FPTwo()
// ========

bits(N) FPTwo(bit sign)
    assert N in {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;
    return result;

Shared Pseudocode Functions
LIBRARY PSEUDOCODE FOR SHARED/FUNCTIONS/FLOAT/FPTYPE/FPTYPE

enumeration FPTYPE {
    FPTYPE_Zero,
    FPTYPE_Denormal,
    FPTYPE_Nonzero,
    FPTYPE_Infinity,
    FPTYPE_QNaN,
    FPTYPE_SNaN};

LIBRARY PSEUDOCODE FOR SHARED/FUNCTIONS/FLOAT/FPUNPACK/FPUNPACK

// FPUnpack()
// =========
(FPTYPE, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
   fpcr.AHP = '0';
   boolean fpexc = TRUE;  // Generate floating-point exceptions
   (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
   return (fp_type, sign, value);

// FPUnpack()
// =========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.
(FPTYPE, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr, boolean fpexc)
   fpcr.AHP = '0';
   (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
   return (fp_type, sign, value);
(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)
  boolean fpexc = TRUE;  // Generate floating-point exceptions
  (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
  return (fp_type, sign, value);

// FPUnpackBase()
// ==============

// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr, boolean fpexc)
  assert N IN {16,32,64};
  boolean altfp = HaveAltFP() && !UsingAArch32();
  boolean fiz = altfp && fpcr.FIZ == '1';
  boolean fz = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
  if N == 16 then
    sign = fpval<15>;
    exp16 = fpval<14:10>;
    frac16 = fpval<9:0>;
    if IsZero(exp16) then
      if IsZero(frac16) || fpcr.FZ16 == '1' then
        fptype = FPType_Zero;  value = 0.0;
      else
        fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
    elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
      fptype = FPType_Infinity;  value = 2.0^1000000;
    else
      fptype = FPType_QNaN else FPType_SNaN;
    value = 0.0;
  else
    fptype = FPType_Nonzero;
    value = 2.0^((UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10));
  elsif N == 32 then
    sign = fpval<31>;
    exp32 = fpval<30:23>;
    frac32 = fpval<22:0>;
    if IsZero(exp32) then
      if IsZero(frac32) then
        // Produce zero if value is zero.
        fptype = FPType_Zero;  value = 0.0;
      elsif fz || fiz then  // Flush-to-zero if FIZ==1 or AH,FZ==01
        fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
      elsif IsOnes(exp32) then
        fptype = FPType_Infinity;  value = 2.0^1000000;
      else
        fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
    else
      fptype = FPType_Nonzero;
      value = 2.0^((UInt(exp32)-23) * (1.0 + Real(UInt(frac32)) * 2.0^-23));
    elsif IsZero(frac32) then
      // Check whether to raise Input Denormal floating-point exception.
      // fpcr.FZ==1 does not raise Input Denormal exception.
      if fz then
        // Denormalized input flushed to zero
        if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
      elsif fpcr.FIZ == '1 then
        // Denormalized input flushed to zero
        if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
      else
        fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
      elsif IsOnes(frac32) then
        fptype = FPTypeInfinity;  value = 2.0^1000000;
      else
        fptype = if fract32<22> == '1' then FPType_QNaN else FPType_SNaN;
value = 0.0;
else
  fptype = FPType_Nonzero;
  value = 2.0^(
    Uint(exp32)-127) * (1.0 + Real(Uint(frac32)) * 2.0^-23);
else // N == 64
  sign = fpval<63>;
  exp64 = fpval<62:52>;
  frac64 = fpval<51:0>;
  if IsZero(exp64) then
    if IsZero(frac64) then
      // Produce zero if value is zero.
      fptype = FPType_Zero; value = 0.0;
    elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
      fptype = FPType_Zero; value = 0.0;
      // Check whether to raise Input Denormal floating-point exception.
      // fpcr.FIZ==1 does not raise Input Denormal exception.
      if fz then
        // Denormalized input flushed to zero
        if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
      else
        fptype = FPType_Denormal; value = 2.0^-1022 * (Real(Uint(frac64)) * 2.0^-52);
      endif
    endif
  else
    if IsOnes(exp64) then
      if IsZero(frac64) then
        fptype = FPType_Infinity; value = 2.0^1000000;
      else
        fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
        value = 0.0;
      endif
    else
      fptype = FPType_Nonzero;
      value = 2.0^(
        Uint(exp64)-1023) * (1.0 + Real(Uint(frac64)) * 2.0^-52);
    endif
    if sign == '1' then value = -value;
  endif
return (fptype, sign, value);

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

// FPUnpackCV()
// ============
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.
(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr)
  fpcr.FZ16 = '0';
  boolean fpexc = TRUE; // Generate floating-point exceptions
  (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
return (fp_type, sign, value);

Library pseudocode for shared/functions/float/fpzero/FPZero

// FPZero()
// ========
bits(N) FPZero(bit sign)
  assert N IN {16,32,64};
  constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
  constant integer F = N - (E + 1);
  exp = Zeros(E);
  frac = Zeros(F);
  result = sign : exp : frac;
return result;
Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;
    return result;

Library pseudocode for shared/functions/integer/AddWithCarry

// AddWithCarry()
// ===============

// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

// AArch64.BranchAddr()
// ===============

// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
Library pseudocode for shared/functions/memory/AccType

```plaintext
enumeration AccType {AccType_NORMAL, AccType_VEC,    // Normal loads and stores
    AccType_STREAM, AccType_VECSTREAM, // Streaming loads and stores
    AccType_ATOMIC, AccType_ATOMICRW,  // Atomic loads and stores
    AccType_ORDERED, AccType_ORDEREDRW, // Load-Acquire and Store-Release
    AccType_ORDEREDATOMIC,    // Load-Acquire and Store-Release with atomic access
    AccType_ORDEREDATOMICRW, // Load-Acquire and Store-Release with atomic access
    AccType_ATOMICLS64,  // Atomic 64-byte loads and stores
    AccType_LIMITEORDERED,  // Load-LOAcquire and Store-LORelease
    AccType_UNPRIV,        // Load and store unprivileged
    AccType_IFETCH,        // Instruction fetch
    AccType_TTW,            // Translation table walk
    AccType_NONFAULT,      // Non-faulting loads
    AccType_CNOTFIRST,     // Contiguous FF load, not first element
    AccType_NV2REGISTER,   // MRS/MSR instruction used at EL1 and which is
                           // to a memory access that uses the EL2 translation
    AccType_CSR_NORMAL,    // Call stack recording, no privilege escalation
    AccType_CSR_PRIV,      // Call stack recording, privilege escalation
    // Other operations
    AccType_DC,            // Data cache maintenance
    AccType_DC_UNPRIV,     // Data cache maintenance instruction used at EL0
    AccType_IC,            // Instruction cache maintenance
    AccType_DCZVA,         // DC ZVA instructions
    AccType_AT};           // Address translation
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```plaintext
type AccessDescriptor is (
    AccType acctype,        // Access type
    boolean transactional,  // Transactional
    MPAMinfo mpam,          // MPAM information
    boolean page_table_walk, // Page table walk
    boolean secondstage,    // Second stage
    boolean s2fs1walk,      // Second stage to first stage
    integer level           // Level
)
```

Library pseudocode for shared/functions/memory/AddrTop

```plaintext
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
assert HaveEL(el);
regime = S1TranslationRegime(el);
if ELUsingAArch32(regime) then
    // AArch32 translation regime.
    return 31;
else
    if EffectiveTBI(address, IsInstr, el) == '1' then
        return 55;
    else
        return 63;
```

Library pseudocode for shared/functions/memory/AddressDescriptor

```plaintext
type AddressDescriptor is (
    FaultRecord fault,    // fault.statusCode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress paddress,
    bits(64) vaddress
)
```
Library pseudocode for shared/functions/memory/Allocation

constant bits(2) MemHint_No = '00'; // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01'; // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10'; // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11'; // Read-Allocate, Write-Allocate

Library pseudocode for shared/functions/memory/BigEndian

// BigEndian()
// ===========
boolean BigEndian(AccType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        return SCTLR_EL2.EE == '1';
    if HaveCSRExt() && acctype == AccType_CSR_PRIV then
        return (IsInHost() && SCTLR_EL2.EE == '1') || (!IsInHost() && SCTLR_EL1.EE == '1');
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;

Library pseudocode for shared/functions/memory/BigEndianReverse

// BigEndianReverse()
// ==================
bits(width) BigEndianReverse (bits(width) value)
assert width IN {8, 16, 32, 64, 128};
integer half = width DIV 2;
if width == 8 then return value;
return
    BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);

Library pseudocode for shared/functions/memory/Cacheability

constant bits(2) MemAttr_NC = '00'; // Non-cacheable
constant bits(2) MemAttr_WT = '10'; // Write-through
constant bits(2) MemAttr_WB = '11'; // Write-back

Library pseudocode for shared/functions/memory/CreateAccessDescriptor

// CreateAccessDescriptor()
// ========================
AccessDescriptor CreateAccessDescriptor(AccType acctype, boolean transactional)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.transactional = transactional;
    accdesc.mpam = GenMPAMcurEL(acctype);
    accdesc.page_table_walk = FALSE;
    return accdesc;

// CreateAccessDescriptor()
// ========================
AccessDescriptor CreateAccessDescriptor(AccType acctype)
    transactional = FALSE;
    return CreateAccessDescriptor(acctype, transactional);
Library pseudocode for shared/functions/memory/CreateAccessDescriptorTTW

// CreateAccessDescriptorTTW()
// ===========================

AccessDescriptor CreateAccessDescriptorTTW(AccType acctype, boolean secondstage, boolean s2fs1walk, integer level)

    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.transactional = FALSE;
    accdesc.mpam = GenMPAMcurEL(acctype);
    accdesc.page_table_walk = TRUE;
    accdesc.s2fs1walk = s2fs1walk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;

Library pseudocode for shared/functions/memory/DataMemoryBarrier

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);

Library pseudocode for shared/functions/memory/DescriptorUpdate
type DescriptorUpdate is (  
    boolean AF,                  // AF needs to be set  
    AddressDescriptor descaddr   // Descriptor to be updated  
)

Library pseudocode for shared/functions/memory/DeviceType

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};

Library pseudocode for shared/functions/memory/EffectiveTBI

// EffectiveTBI()  
// ==============  
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACExt() then
tbid = if address<55> == '1' then TCR_EL1.TBD1 else TCR_EL1.TBD0;
        when EL2
            if HaveVirtHostExt() & ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if HavePACExt() then
tbid = if address<55> == '1' then TCR_EL2.TBD1 else TCR_EL2.TBD0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACExt() then tbid = TCR_EL2.TBD;
        when EL3
            tbi = TCR_EL3.TBI;
            if HavePACExt() then tbid = TCR_EL3.TBD;
        else
            tbi = '1';
            if HavePACExt() then tbid = '0';
    return (if tbi == '1' & (HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');
Library pseudocode for shared/functions/memory/EffectiveTCMA

// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
assert HaveEL(el);
regime = S1TranslationRegime(el);
assert(!ELUsingAArch32(regime));
case regime of
  when EL1
    tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
  when EL2
    if HaveVirtHostExt() && ELIsInHost(el) then
      tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
    else
      tcma = TCR_EL2.TCMA;
  when EL3
    tcma = TCR_EL3.TCMA;
return tcma;

Library pseudocode for shared/functions/memory/Fault

enumeration Fault {Fault_None,
  Fault_AccessFlag,
  Fault_Alignment,
  Fault_Background,
  Fault_Domain,
  Fault_Permission,
  Fault_Translation,
  Fault_AddressSize,
  Fault_SyncExternal,
  Fault_SyncExternalOnWalk,
  Fault_SyncParity,
  Fault_SyncParityOnWalk,
  Fault_AsyncParity,
  Fault_AsyncExternal,
  Fault_Debug,
  Fault_TLBConflict,
  Fault_BranchTarget,
  Fault_HWUpdateAccessFlag,
  Fault_Lockdown,
  Fault_Thermal,
  Fault_Exclusive,
  Fault_ICacheMaint};
Library pseudocode for shared/functions/memory/FaultRecord

type FaultRecord is (Fault statuscode, // Fault Status
AccType acctype, // Type of access that faulted
FullAddress ipaddress, // Intermediate physical address
boolean s2fs1walk, // Is on a Stage 1 translation table walk
boolean write, // TRUE for a write, FALSE for a read
integer level, // For translation, access flag and permission faults
bit extflag, // IMPLEMENTATION DEFINED syndrome for external aborts
bit(4) domain, // Domain number, AArch32 only
bit(2) errortype, // [Armv8.2 RAS] AArch32 AET or AArch64 SET
bit(4) debugmoe // Debug method of entry, from AArch32 only)

type PARTIDtype = bits(16);
type PMGtype = bits(8);
type MPAMinfo is (
    bit mpam_ns,
    PARTIDtype partid,
    PMGtype pmg)

Library pseudocode for shared/functions/memory/FullAddress

type FullAddress is (bits(52) address,
    bit NS // '0' = Secure, '1' = Non-secure)

Library pseudocode for shared/functions/memory/Hint_Prefetch

// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);

Library pseudocode for shared/functions/memory/IsDataAccess

// IsDataAccess()
// ==============
// Return TRUE if access is to data memory.

boolean IsDataAccess(AccType acctype)
    return !(acctype IN {AccType_IFETCH, AccType_TTW,
        AccType_DC, AccType_DC_UNPRIV,
        AccType_IC, AccType_AT});

Library pseudocode for shared/functions/memory/MBReqDomain

enumeration MBReqDomain {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
    MBReqDomain_OuterShareable, MBReqDomain_FullSystem};

Library pseudocode for shared/functions/memory/MBReqTypes

enumeration MBReqTypes {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
Library pseudocode for shared/functions/memory/MemAttrHints

type MemAttrHints is (  
    bits(2) attrs, // See MemAttr_*, Cacheability attributes  
    bits(2) hints, // See MemHint_*, Allocation hints  
    boolean transient  
)

Library pseudocode for shared/functions/memory/MemType

enumeration MemType {MemType_Normal, MemType_Device};

Library pseudocode for shared/functions/memory/MemoryAttributes

type MemoryAttributes is (  
    MemType memtype,  
    DeviceType device, // For Device memory types  
    MemAttrHints inner, // Inner hints and attributes  
    MemAttrHints outer, // Outer hints and attributes  
    boolean tagged, // Tagged access  
    boolean shareable,  
    boolean outershareable  
)

Library pseudocode for shared/functions/memory/Permissions

type Permissions is (  
    bits(3) ap, // Access permission bits  
    bit xn, // Execute-never bit  
    bit xxn, // [Armv8.2] Extended execute-never bit for stage 2  
    bit pxn // Privileged execute-never bit  
)

Library pseudocode for shared/functions/memory/PrefetchHint

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

SpeculativeStoreBypassBarrierToPA();

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

SpeculativeStoreBypassBarrierToVA();

Library pseudocode for shared/functions/memory/TLBRecord

type TLBRecord is (  
    Permissions perms, // '0' = Global, '1' = not Global  
    bit ng, // Guarded Page  
    bits(4) domain, // AArch32 only  
    bit GP, // AArch32 Short-descriptor format: Indicates Section/Page  
    boolean contiguous, // Contiguous bit from translation table  
    integer level, // AArch32 Short-descriptor format: Indicates Section/Page  
    integer blocksize, // Describes size of memory translated in KBytes  
    DescriptorUpdate descupdate, // [Armv8.1] Context for h/w update of table descriptor  
    bit CnP, // [Armv8.2] TLB entry can be shared between different PEs  
    AddressDescriptor addrdesc  
)
constant integer LOG2_TAG_GRANULE = 4;
constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;

// These two Mem[] accessors are the hardware operations which perform single-copy atomic,
// aligned, little-endian memory accesses of size bytes from/to the underlying physical
// memory array of bytes.
//
// The functions address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an external abort.
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc] = bits(8*size) value;

// Returns default MPAM info. If secure is TRUE return default Secure
// MPAMinfo, otherwise return default Non-secure MPAMinfo.
MPAMinfo DefaultMPAMinfo(boolean secure)
    MPAMinfo DefaultInfo;
    DefaultInfo.mpam_ns = if secure then '0' else '1';
    DefaultInfo.partid = DefaultPARTID;
    DefaultInfo.pmg = DefaultPMG;
    return DefaultInfo;

constant PARTIDtype DefaultPARTID = 0<15:0>;
constant PMGtype DefaultPMG = 0<7:0>;
/ Library pseudocode for shared/functions/mpam/GenMPAMcurEL

// GenMPAMcurEL
// ============
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32. In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation. If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccType acctype)
bits(2) mpamel;
boolean validEL;
boolean securempam;
boolean InD = acctype IN {AccType_IFETCH, AccType_IC};
if HaveEMPAMExt() then
  boolean secure = IsSecure();
  securempam = MPAM3_EL3.FORCE_NS == '0' && secure;
  if MPAMisEnabled() && (!secure || MPAM3_EL3.SDEFLT == '0') then
    if UsingAArch32() then
      (validEL, mpamel) = ELFromM32(PSTATE.M);
    else
      validEL = TRUE;
    if acctype == AccType_CSR_PRIV then
      assert HaveCSRExt();
      mpamel = if IsInHost() then EL2 else EL1;
    else
      mpamel = PSTATE.EL;
    if validEL then
      return genMPAM(UInt(mpamel), InD, securempam);
  else
    securempam = IsSecure();
    if HaveMPAMExt() & MPAMisEnabled() then
      if UsingAArch32() then
        (validEL, mpamel) = ELFromM32(PSTATE.M);
      else
        validEL = TRUE;
        if acctype == AccType_CSR_PRIV then
          mpamel = if IsInHost() then EL2 else EL1;
        else
          mpamel = PSTATE.EL;
        if validEL then
          return genMPAM(UInt(mpamel), InD, securempam);
    return DefaultMPAMinfo(securempam);
Library pseudocode for shared/functions/mpam/MAP_vPARTID

// MAP_vPARTID
// ===========
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
// should not ever be called if EL2 is not implemented
// or is implemented but not enabled in the current
// security state.
PARTIDtype ret;
boolean err;
integer virt = UInt( vpartid );
integer vpmrmax = UInt( MPAMIDR_EL1.VPMR_MAX );

// vpartid_max is largest vpartid supported
integer vpartid_max = (4 * vpmrmax) + 3;

// One of many ways to reduce vpartid to value less than vpartid_max.
if virt > vpartid_max then
  virt = virt MOD (vpartid_max+1);

// Check for valid mapping entry.
if MPAMVPMV_EL2<virt> == '1' then
  // vpartid has a valid mapping so access the map.
  ret = mapvpmw(virt);
  err = FALSE;

// Is the default virtual PARTID valid?
elsf MPAMVPMV_EL2<0> == '1' then
  // Yes, so use default mapping for vpartid == 0.
  ret = MPAMVPM0_EL2<0 : 16>;
  err = FALSE;

// Neither is valid so use default physical PARTID.
else
  ret = DefaultPARTID;
  err = TRUE;

// Check that the physical PARTID is in-range.
// This physical PARTID came from a virtual mapping entry.
integer partid_max = UInt( MPAMIDR_EL1.PARTID_MAX );
if UInt(ret) > partid_max then
  // Out of range, so return default physical PARTID
  ret = DefaultPARTID;
  err = TRUE;
return (ret, err);

Library pseudocode for shared/functions/mpam/MPAMisEnabled

// MPAMisEnabled
// ============
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
  el = HighestEL();
case el of
  when EL3 return MPAM3_EL3.MPAMEN == '1';
  when EL2 return MPAM2_EL2.MPAMEN == '1';
  when EL1 return MPAM1_EL1.MPAMEN == '1';
// MPAMisVirtual
// =============
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(integer el) {
    return ( MPAMIDR_EL1.HAS_HCR == '1' &&
            EL2Enabled() &&
            (( el == 0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
                ( HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0' )) ||
            ( el == 1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
}

// genMPAM
// =======
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// Produces a Secure PARTID if Secure is TRUE and a Non-secure PARTID otherwise.

MPAMinfo genMPAM(integer el, boolean InD, boolean secure) {
    MPAMinfo returnInfo;
    PARTIDtype partidel;
    boolean perr;
    boolean gstplk = (el == 0 && EL2Enabled() &&
                      MPAMHCR_EL2.GSTAPP_PLK == '1' && HCR_EL2.TGE == '0');
    integer eff_el = if gstplk then 1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel = genPMG(eff_el, InD, perr);
    returnInfo.mpam_ns = if secure then '0' else '1';
    returnInfo.partid  = partidel;
    returnInfo.pmg     = groupel;
    return returnInfo;
}

// genMPAMel
// ==========
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

MPAMinfo genMPAMel(bits(2) el, boolean InD) {
    boolean secure = IsSecure();
    boolean securempam = secure;
    if HaveMPAMExt() then
        securempam = MPAM3_EL3.FORCE_NS == '0' && secure;
        if HaveMPAMExt() && MPAMisEnabled() && (!secure || MPAM3_EL3.SDEFLT == '0') then
            return genMPAM(UInt(el), InD, securempam);
    else
        if HaveMPAMExt() && MPAMisEnabled() then
            return genMPAM(UInt(el), InD, securempam);
        return DefaultMPAMinfo(securempam);
    return returnInfo;
}
// genPARTID
// ========
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.

(PARTIDtype, boolean) genPARTID(integer el, boolean InD)

PARTIDtype partidel = getMPAM_PARTID(el, InD);

integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
if UInt(partidel) > partid_max then
    return (DefaultPARTID, TRUE);

if MPAMisVirtual(el) then
    return MAP_vPARTID(partidel);
else
    return (partidel, FALSE);

// genPMG
// ======
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(integer el, boolean InD, boolean partid_err)

integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
if partid_err then
    return DefaultPMG;

PMGtype groupel = getMPAM_PMG(el, InD);
if UInt(groupel) <= pmg_max then
    return groupel;
else
    return DefaultPMG;

// getMPAM_PARTID
// ==============
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(integer MPAMn, boolean InD)

PARTIDtype partid;
boolean el2avail = EL2Enabled();

if InD then
    case MPAMn of
        when 3 partid = MPAM3_EL3.PARTID_I;
        when 2 partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
        when 1 partid = MPAM1_EL1.PARTID_I;
        when 0 partid = MPAM0_EL1.PARTID_I;
    otherwise partid = PARTIDtype UNKNOWN;
else
    case MPAMn of
        when 3 partid = MPAM3_EL3.PARTID_D;
        when 2 partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
        when 1 partid = MPAM1_EL1.PARTID_D;
        when 0 partid = MPAM0_EL1.PARTID_D;
    otherwise partid = PARTIDtype UNKNOWN;

return partid;
Library pseudocode for shared/functions/mpam/getMPAM_PMG

// getMPAM_PMG
// ===========
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(integer MPAMn, boolean InD)
{
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when 3 pmg = MPAM3_EL3.PMG_I;
            when 2 pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
            when 1 pmg = MPAM1_EL1.PMG_I;
            when 0 pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
    else
        case MPAMn of
            when 3 pmg = MPAM3_EL3.PMG_D;
            when 2 pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros();
            when 1 pmg = MPAM1_EL1.PMG_D;
            when 0 pmg = MPAM0_EL1.PMG_D;
            otherwise pmg = PMGtype UNKNOWN;
    return pmg;
}

Library pseudocode for shared/functions/mpam/mapvpmw

// mapvpmw
// =======
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
{
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb +: 16>;
}
Library pseudocode for shared/functions/registers/BranchTo

// BranchTo()
// ===========

// Set program counter to a new address, with a branch type
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type) {
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        BranchRecord(branch_type, target<63:0>);
        _PC = AArch64.BranchAddr(target<63:0>);
    return;
}

Library pseudocode for shared/functions/registers/BranchToAddr

// BranchToAddr()
// ==============

// Set program counter to a new address, with a branch type
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type) {
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
    else
        assert N == 64 && !UsingAArch32();
        BranchRecord(branch_type, target<63:0>);
        _PC = target<63:0>;
    return;
}

Library pseudocode for shared/functions/registers/BranchType

enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
    BranchType_DIR,         // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_TMFAIL,      // Transaction failure
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};    // Other

Library pseudocode for shared/functions/registers/Hint_Branch

// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);

Library pseudocode for shared/functions/registers/NextInstrAddr

// Return address of the sequentially next instruction.
bits(N) NextInstrAddr();
Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);

Library pseudocode for shared/functions/registers/ThisInstrAddr

// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
assert N == 64 || (N == 32 & UsingAArch32());
return _PC<N-1:0>;

Library pseudocode for shared/functions/registers/_PC

bits(64) _PC;

Library pseudocode for shared/functions/registers/_R

array bits(64) _R[0..30];
Library pseudocode for shared/functions/sysregisters/SPSR

// SPSR[] - non-assignment form
// -------------------------------

bits(N) SPSR[]
    bits(N) result;
    if UsingAArch32() then
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ       result = SPSR_fiq\langle N-1:0\rangle;
            when M32_IRQ       result = SPSR_irq\langle N-1:0\rangle;
            when M32_Svc       result = SPSR_svc\langle N-1:0\rangle;
            when M32_Monitor   result = SPSR_mon\langle N-1:0\rangle;
            when M32_Abort     result = SPSR_abt\langle N-1:0\rangle;
            when M32_Hyp       result = SPSR_hyp\langle N-1:0\rangle;
            when M32_Undef     result = SPSR_und\langle N-1:0\rangle;
            otherwise         Unreachable();
        else
            assert N == 64;
            case PSTATE.EL of
                when EL1         result = SPSR_EL1\langle N-1:0\rangle;
                when EL2         result = SPSR_EL2\langle N-1:0\rangle;
                when EL3         result = SPSR_EL3\langle N-1:0\rangle;
                otherwise       Unreachable();
            return result;
        end
    end
// SPSR[] - assignment form
// -------------------------------

SPSR[] = bits(N) value
if UsingAArch32() then
    assert N == 32;
    case PSTATE.M of
        when M32_FIQ       SPSR_fiq = ZeroExtend(value);
        when M32_IRQ       SPSR_irq = ZeroExtend(value);
        when M32_Svc       SPSR_svc = ZeroExtend(value);
        when M32_Monitor   SPSR_mon = ZeroExtend(value);
        when M32_Abort     SPSR_abt = ZeroExtend(value);
        when M32_Hyp       SPSR_hyp = ZeroExtend(value);
        when M32_Undef     SPSR_und = ZeroExtend(value);
        otherwise         Unreachable();
    else
        assert N == 64;
        case PSTATE.EL of
            when EL1         SPSR_EL1 = ZeroExtend(value);
            when EL2         SPSR_EL2 = ZeroExtend(value);
            when EL3         SPSR_EL3 = ZeroExtend(value);
            otherwise       Unreachable();
        return;
    end

// Library pseudocode for shared/functions/system/ArchVersion

enumeration ArchVersion {
    ARMv8p0
    , ARMv8p1
    , ARMv8p2
    , ARMv8p3
    , ARMv8p4
    , ARMv8p5
    , ARMv8p6
    , ARMv8p7
};
Library pseudocode for shared/functions/system/BranchTargetCheck

// BranchTargetCheck()
// ===============
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
assert HaveBTIExt() && !UsingAArch32();

// The branch target check considers two state variables:
// * InGuardedPage, which is evaluated during instruction fetch.
// * BTypeCompatible, which is evaluated during instruction decode.
if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
    bits(64) pc = ThisInstrAddr();
    AArch64.BranchTargetException(pc<51:0>);

boolean branch_instr = AArch64.ExecutingBRORBLRORRetInstr();
boolean bti_instr    = AArch64.ExecutingBTIIInstr();

// PSTATE.BTYPE defaults to 00 for instructions that do not explicitly set BTYPE.
if !(branch_instr || bti_instr) then
    BTypeNext = '00';

Library pseudocode for shared/functions/system/ClearEventRegister

// ClearEventRegister()
// ===============
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;

Library pseudocode for shared/functions/system/ClearPendingPhysicalSError

// Clear a pending physical SError interrupt.
ClearPendingPhysicalSError();

Library pseudocode for shared/functions/system/ClearPendingVirtualSError

// Clear a pending virtual SError interrupt.
ClearPendingVirtualSError();
// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');  // EQ or NE
        when '001' result = (PSTATE.C == '1');  // CS or CC
        when '010' result = (PSTATE.N == '1');  // MI or PL
        when '011' result = (PSTATE.V == '1');  // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');  // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);  // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' result = TRUE;  // AL
    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;
    return result;

ConsumptionOfSpeculativeDataBarrier();

// CurrentInstrSet()
// =============
InstrSet CurrentInstrSet()
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);

constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
Library pseudocode for shared/functions/system/EL2Enabled

```java
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with SCR_EL3.NS==1 when Non-secure EL2 is implemented, or
// - with SCR_EL3.NS==0 when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_EL3.NS == '1' || IsSecureEL2Enabled());
```

Library pseudocode for shared/functions/system/ELFromM32

```java
// ELFromM32()
// ===========
// Convert an AArch32 mode encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
//       and the current value of SCR.NS/SCR_EL3.NS.
//   'EL'    is the Exception level decoded from 'mode'.
// bits(2) el;
// boolean valid = !BadMode(mode); // Check for modes that are not valid for this implementation
// case mode of
//     when M32_Monitor
//         el = EL3;
//     when M32_Hyp
//         el = EL2;
//         valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
//     when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32.Undef, M32_System
//         // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
//         // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
//         // AArch64, then these modes are EL1 modes.
//         el = (if HaveEL(EL3) && HighestELUsingAArch32() && SCR.NS == '0' then EL3 else EL1);
//     when M32_User
//         el = EL0;
//     otherwise
//         valid = FALSE; // Passed an illegal mode value
// if !valid then el = bits(2) UNKNOWN;
// return (valid, el);
```
Library pseudocode for shared/functions/system/ELFromSPSR

// ELFromSPSR()
// ============
// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
// 'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
// 'EL' is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(N) spsr)
if spsr<4> == '0' then                      // AArch64 state
  el = spsr<3:2>;
elseif !HighestELUsingAArch32() then         // No AArch64 support
  valid = FALSE;
elsif !HaveEL(el) then                  // Exception level not implemented
  valid = FALSE;
elsif spsr<1> == '1' then               // M[1] must be 0
  valid = FALSE;
elsif el == EL0 && spsr<0> == '1' then  // for EL0, M[0] must be 0
  valid = FALSE;
elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
  valid = FALSE;                      // Unless Secure EL2 is enabled, EL2 only valid in Non-secure state
else
  valid = TRUE;
elsif HaveAnyAArch32() then                // AArch32 state
  (valid, el) = ELFromM32(spsr<4:0>);
else
  valid = FALSE;
if !valid then el = bits(2) UNKNOWN;
return (valid,el);

Library pseudocode for shared/functions/system/ELIsInHost

// ELIsInHost()
// ============
boolean ELIsInHost(bits(2) el)
if !HaveVirtHostExt() || ELUsingAArch32(EL2) then
  return FALSE;
else
  case el of
    when EL3
      return FALSE;
    when EL2
      return HCR_EL2.E2H == '1';
    when EL1
      return FALSE;
    when EL0
      return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
    otherwise
      Unreachable();
  end

Library pseudocode for shared/functions/system/ELStateUsingAArch32

// ELStateUsingAArch32()
// ================
boolean ELStateUsingAArch32(bits(2) el, boolean secure)
// See ELStateUsingAArch32K() for description. Must only be called in circumstances where
// result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
(known, aarch32) = ELStateUsingAArch32K(el, secure);
assert known;
return aarch32;
Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```plaintext
(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure) // Returns (known, aarch32):
  // 'known' is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
  // using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
  // 'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
  if !HaveAArch32EL(el) then
    return (TRUE, FALSE);  // Exception level is using AArch64
  elsif secure && el == EL2 then
    return (TRUE, FALSE);  // Secure EL2 is using AArch64
  elsif HighestELUsingAArch32() then
    return (TRUE, TRUE);   // Highest Exception level, and therefore all levels are
                           // using AArch64
  elsif el == HighestEL() then
    return (TRUE, FALSE);  // This is highest Exception level, so is using AArch64
  else
    boolean aarch32 = boolean UNKNOWN;
    boolean known = TRUE;
    aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext());
    aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) &&
                       (HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) && HCR_EL2.RW == '0' &&
                       !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt()));
    if el == EL0 && !aarch32_at_el1 then       // Only know if EL0 using AArch32 from PSTATE
      if PSTATE.EL == EL0 then
        aarch32 = PSTATE.nRW == '1';       // EL0 controlled by PSTATE
      else
        known = FALSE;                     // EL0 state is UNKNOWN
      end
    else
      aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});
    end
    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

Library pseudocode for shared/functions/system/ELUsingAArch32

```plaintext
// ELUsingAArch32()
// ===============

boolean ELUsingAArch32(bits(2) el)
return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/ELUsingAArch32K

```plaintext
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/EndOfInstruction

```plaintext
// Terminate processing of the current instruction.
EndOfInstruction();
```

Library pseudocode for shared/functions/system/EnterLowPowerState

```plaintext
// PE enters a low-power state.
EnterLowPowerState();
```
Library pseudocode for shared/functions/system/EventRegister

bits(1) EventRegister;

Library pseudocode for shared/functions/system/ExceptionalOccurrenceTargetState

enumeration ExceptionalOccurrenceTargetState {
  AArch32_NonDebugState,
  AArch64_NonDebugState,
  DebugState
};

Library pseudocode for shared/functions/system/FIQPending

// Returns TRUE if there is any pending physical FIQ
boolean FIQPending();

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState)
  if UsingAArch32() & (targetELState IN {AArch32_NonDebugState, DebugState}) then
    assert N == 32;
  else
    assert N == 64;
  bits(N) spsr = Zeros();
  spsr<31:28> = PSTATE.<N,Z,C,V>;
  if HavePANExt() then spsr<22> = PSTATE.PAN;
  spsr<20> = PSTATE.IL;
  if PSTATE.nRW == '1' then // AArch32 state
    spsr<27> = PSTATE.Q;
    spsr<26:25> = PSTATE.IT<1:0>;
    if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
    if HaveDITExt() then
      if targetELState == AArch32_NonDebugState then
        spsr<21> = PSTATE.DIT;
      else // AArch64_NonDebugState or DebugState
        spsr<21> = PSTATE.DIT;
    if targetELState IN {AArch64_NonDebugState, DebugState} then
      spsr<21> = PSTATE.DIT;
    if targetELState IN {AArch64_NonDebugState, DebugState} then
      spsr<21> = PSTATE.DIT;
  else
    if HaveCSRExt() then spsr<32> = PSTATE.CSR;
    if HaveMTEExt() then spsr<25> = PSTATE.TCO;
    if HaveDITExt() then spsr<24> = PSTATE.DIT;
    if HaveUAOExt() then spsr<23> = PSTATE.UAO;
    spsr<21> = PSTATE.SS;
    if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
    if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
    spsr<9:6> = PSTATE.<D,A,I,F>;
    spsr<4> = PSTATE.nRW;
    spsr<3:2> = PSTATE.EL;
    spsr<0> = PSTATE.SP;
  return spsr;
Library pseudocode for shared/functions/system/HasArchVersion

// HasArchVersion()
// ================
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/system/HaveAArch32EL

// HaveAArch32EL()
// =============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;         // The Exception level is not implemented
    elsif !HaveAnyAArch32() then
        return FALSE;         // No Exception level can use AArch32
    elsif HighestELUsingAArch32() then
        return TRUE;          // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;         // The highest Exception level is using AArch64
    elsif el == EL0 then
        return TRUE;          // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/system/HaveAnyAArch32

// HaveAnyAArch32()
// ===============

boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/system/HaveAnyAArch64

// HaveAnyAArch64()
// ===============

boolean HaveAnyAArch64()
    return !HighestELUsingAArch32();

Library pseudocode for shared/functions/system/HaveEL

// HaveEL()
// ========

boolean HaveEL(bits(2) el)
    if el IN {EL1, EL0} then
        return TRUE;          // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
Library pseudocode for shared/functions/system/HaveELUsingSecurityState

// HaveELUsingSecurityState()
// =========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)
{
    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && HaveSecureEL2Ext();
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
}

Library pseudocode for shared/functions/system/HaveFP16Ext

// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
{
    return boolean IMPLEMENTATION_DEFINED;
}

Library pseudocode for shared/functions/system/HighestEL

// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
{
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
}

Library pseudocode for shared/functions/system/HighestELUsingAArch32

// HighestELUsingAArch32()
// =======================
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
{
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED;       // e.g. CFG32SIGNAL == HIGH
}

Library pseudocode for shared/functions/system/Hint_DGH

// Provides a hint to close any gathering occurring within the micro-architecture.
Hint_DGH();
Library pseudocode for shared/functions/system/Hint_WFE

// Hint_WFE()
// =========
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

Hint_WFE(integer localtimeout)
    if IsEventRegisterSet() then
        ClearEventRegister();
    else
        trap = FALSE;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            if HaveTWEDExt() then
                sctlr = SCTLR[];
                trap = sctlr.nTWE == '0';
                target_el = EL1;
            else
                AArch64.CheckForWFxTrap(EL1, TRUE);
            end
        else
            AArch64.CheckForWFxTrap(EL2, TRUE);
        end
        if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            if HaveTWEDExt() then
                trap = HCR_EL2.TWE == '1';
                target_el = EL2;
            else
                AArch64.CheckForWFxTrap(EL2, TRUE);
            end
        else
            AArch64.CheckForWFxTrap(EL3, TRUE);
        end
        if !trap && HaveEl(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            if HaveTWEDExt() then
                trap = SCR_EL3.TWE == '1';
                target_el = EL3;
            else
                AArch64.CheckForWFxTrap(EL3, TRUE);
            end
        else
            AArch64.CheckForWFxTrap(EL3, TRUE);
        end
        if trap && PSTATE.EL != EL3 then
            (delay_enabled, delay) = WFETrapDelay(target_el);   // (If trap delay is enabled, Delay amount)
            if !WaitForEventUntilDelay(delay_enabled, delay) then
                // Event did not arrive before delay expired
                AArch64.WFxTrap(target_el, TRUE);                // Trap WFE
            end
        else
            WaitForEvent(localtimeout);
Library pseudocode for shared/functions/system/Hint_WFI

// Hint_WFI()
// =========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFI(integer localtimeout)
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);
    if !InterruptPending() then
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            AArch64.CheckForWFxTrap(EL1, FALSE);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFxTrap(EL2, FALSE);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL3, FALSE);
        WaitForInterrupt(localtimeout);

Library pseudocode for shared/functions/system/Hint_Yield

// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.
Hint_Yield();

Library pseudocode for shared/functions/system/IRQPending

// Returns TRUE if there is any pending physical IRQ
boolean IRQPending();
Library pseudocode for shared/functions/system/IllegalExceptionReturn

```java
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(N) spsr)
{
    // Check for illegal return:
    // * To an unimplemented Exception level.
    // * To EL2 in Secure state, when SecureEL2 is not enabled.
    // * To EL0 using AArch64 state, with SPSR.M[0]==1.
    // * To AArch64 state with SPSR.M[1]==1.
    // * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    // * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //   Execution state used in the Exception level being returned to, as determined by
    //   the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    // * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //   SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    // * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
}
```

Library pseudocode for shared/functions/system/InstrSet

```java
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```java
InstructionSynchronizationBarrier();
```

Library pseudocode for shared/functions/system/InterruptPending

```java
// InterruptPending()
// ===============
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
{
    bit vIRQstatus = (if VirtualIRQPending() then '1' else '0') OR HCR_EL2.VI;
    bit vFIQstatus = (if VirtualFIQPending() then '1' else '0') OR HCR_EL2.VF;
    bits(3) v_interrupts = HCR_EL2.VSE : vIRQstatus : vFIQstatus;

    pending_physical_interrupt = (IRQPending() || FIQPending() ||
                                  IsPhysicalSErrorPending());
    pending_virtual_interrupt = !IsInHost() && ((v_interrupts AND
                                            HCR_EL2.<AMO,IMO,FMO>) != '000');

    return pending_physical_interrupt || pending_virtual_interrupt;
}
Library pseudocode for shared/functions/system/IsEventRegisterSet

// IsEventRegisterSet()
//=---------------------
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise.

boolean IsEventRegisterSet()
    return EventRegister == '1';

Library pseudocode for shared/functions/system/IsHighestEL

// IsHighestEL()
//=-------------
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;

Library pseudocode for shared/functions/system/IsInHost

// IsInHost()
//=---------

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);

Library pseudocode for shared/functions/system/IsPhysicalSErrorPending

// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();

Library pseudocode for shared/functions/system/IsSErrorEdgeTriggered

// IsSErrorEdgeTriggered()
//=----------------------
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered(bits(24) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;
        if UsingAArch32() && syndrome<11:10> != '00' then
            // AArch32 and not Uncontainable.
            return TRUE;
        if !UsingAArch32() && syndrome<23> == '0' && syndrome<5:0> != '000000' then
            // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor Uncategorized.
            return TRUE;
        return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";

Library pseudocode for shared/functions/system/IsSecure

// IsSecure()
//=---------
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL(EL3) && !UsingAArch32() && PSTATE_EL == EL3 then
        return TRUE;
    elsif HaveEL(EL3) && UsingAArch32() && PSTATE_M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
Library pseudocode for shared/functions/system/IsSecureBelowEL3

// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
// //
// // Differs from IsSecure in that it ignores the current EL or Mode
// // in considering security state.
// // That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// // exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()

if HaveEL(EL3) then
    return SCR_GEN[].NS == '0';
elsif HaveEL(EL2) && (!HaveSecureEL2Ext() || HighestELUsingAArch32()) then
    // If Secure EL2 is not an architecture option then we must be Non-secure.
    return FALSE;
else
    // TRUE if processor is Secure or FALSE if Non-secure.
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";

Library pseudocode for shared/functions/system/IsSecureEL2Enabled

// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()

if HaveEL(EL2) && HaveSecureEL2Ext() then
    if HaveEL(EL3) then
        if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
            return TRUE;
        else
            return FALSE;
    else
        return IsSecure();
else
    return FALSE;

Library pseudocode for shared/functions/system/IsSynchronizablePhysicalSErrorPending

// Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsSynchronizablePhysicalSErrorPending();

Library pseudocode for shared/functions/system/IsVirtualSErrorPending

// Returns TRUE if a virtual SError interrupt is pending.
boolean IsVirtualSErrorPending();

Library pseudocode for shared/functions/system/LocalTimeoutEvent

// Returns TRUE if a local timeout event is generated when the value of
// CNTVCT_EL0 equals or exceeds the threshold value for the first time.
// If the threshold value is less than zero a local timeout event will
// not be generated.
boolean LocalTimeoutEvent(integer localtimeout);
constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';

// PLOfEL()
// ========
PrivilegeLevel PLOfEL(bits(2) el)
case el of
  when EL3 return if HighestELUsingAArch32() then PL1 else PL3;
  when EL2 return PL2;
  when EL1 return PL1;
  when EL0 return PL0;

// PSTATE
ProcState PSTATE;

// PrivilegeLevel
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};

// ProcState
type ProcState is (  
  bits (1) N,        // Negative condition flag  
  bits (1) Z,        // Zero condition flag  
  bits (1) C,        // Carry condition flag  
  bits (1) V,        // oVerflow condition flag  
  bits (1) D,        // Debug mask bit                     [AArch64 only]
  bits (1) A,        // SError interrupt mask bit  
  bits (1) I,        // IRQ mask bit  
  bits (1) F,        // FIQ mask bit  
  bits (1) PAN,      // Privileged Access Never Bit [v8.1]
  bits (1) UAO,      // User Access Override   [v8.2]
  bits (1) DIT,      // Data Independent Timing [v8.4]
  bits (1) TCO,      // Tag Check Override  [v8.5, AArch64 only]
  bits (1) CSR,      // Call Stack Recording  
  bits (2) BTYPE,    // Branch Type                     [v8.5]
  bits (1) SS,       // Software step bit  
  bits (1) IL,       // Illegal Execution state bit  
  bits (2) EL,       // Exception Level  
  bits (1) nRW,      // not Register Width: 0=64, 1=32  
  bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
  bits (1) Q,        // Cumulative saturation flag [AArch32 only]
  bits (4) GE,       // Greater than or Equal flags  [AArch32 only]
  bits (1) SSBS,     // Speculative Store Bypass Safe  
  bits (8) IT,       // If-then bits, RES0 in CPSR [AArch32 only]
  bits (1) J,        // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]
  bits (1) T,        // T32 bit, RES0 in CPSR [AArch32 only]
  bits (1) E,        // Endianness bit [AArch32 only]
  bits (5) M         // Mode field [AArch32 only]  
);
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
    it = spsr<15:10,26:25>;
    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;
    // The IT bits are forced to zero when they are set to a reserved value.
    if IsZero(it<7:4>) && IsZero(it<3:0>) then return '00000000';
    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;

Library pseudocode for shared/functions/system/SCRType

type SCRTypen;
SetPSTATEFromPSR(bits(N) spsr)
    boolean from_aarch64 = !UsingAArch32();
    assert N == (if from_aarch64 then 64 else 32);
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    ShouldAdvanceSS = FALSE;
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIEExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITEExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = bit UNKNOWN;
        if HaveCSRExt() then PSTATE.CSR = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then // AArch32 state
            AArch32.WriteMode(spsr<4:0>); // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;
            PSTATE.SP = spsr<0>;
            if HaveBTIEExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
            if HaveCSRExt() then PSTATE.CSR = spsr<32>;
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveDITEExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEExt() then PSTATE.TCO = spsr<25>;
    // If PSTATE.IL is set and returning to AArch32 state, it is CONSTRAINED UNPREDICTABLE whether
    // the T bit is set to zero or copied from SPR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstraintUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';
    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if HavePANExt() then PSTATE.PAN = spsr<22>;
    if PSTATE.nRW == '1' then // AArch32 state
        PSTATE.GE = spsr<19:16>;
        PSTATE.E = spsr<9>;
        PSTATE.<A,I,F> = spsr<8:6>;
    else // AArch64 state
        PSTATE.<D,A,I,F> = spsr<9:6>;
    return;

shouldAdvanceIT();

shouldAdvanceSS();

SpeculationBarrier();
SynchronizeContext();

// Implements the error synchronization event.
SynchronizeErrors();

// Take any pending unmasked physical SError interrupt or unmasked virtual SError interrupt.
TakeUnmaskedSErrorInterrupts();

bits(32) ThisInstr();

integer ThisInstrLength();

assert FALSE;

boolean UsingAArch32()

// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean aarch32 = (PSTATE.nRW == '1');
if !HaveAnyAArch32() then assert !aarch32;
if HighestELUsingAArch32() then assert aarch32;
return aarch32;

boolean VirtualFIQPending();

boolean VirtualIRQPending();

// Returns TRUE if there is any pending virtual FIQ
bool VirtualFIQPending();

// Returns TRUE if there is any pending virtual IRQ
bool VirtualIRQPending();

Library pseudocode for shared/functions/system/VirtualIRQPending

Library pseudocode for shared/functions/system/SynchronizeContext

Library pseudocode for shared/functions/system/SynchronizeErrors

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

Library pseudocode for shared/functions/system/ThisInstr

Library pseudocode for shared/functions/system/ThisInstrLength

Library pseudocode for shared/functions/system/Unreachable

Library pseudocode for shared/functions/system/UsingAArch32

Library pseudocode for shared/functions/system/VirtualFIQPending

Library pseudocode for shared/functions/system/VirtualIRQPending
Library pseudocode for shared/functions/system/WaitForEvent

// WaitForEvent()
// ==============
// PE suspends its operation and enters a low-power state if the
// Event Register is clear and, for WFET, there is no Local
// Timeout event when the WFET is executed.

WaitForEvent(integer localtimeout)
    if !(IsEventRegisterSet() || LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    return;

Library pseudocode for shared/functions/system/WaitForInterrupt

// WaitForInterrupt()
// ==============
// PE suspends its operation to enter a low-power state until
// a WFI wake-up event occurs, the PE is reset, and, for WFIT,
// a Local Timeout Event is generated.

WaitForInterrupt(integer localtimeout)
    if localtimeout < 0 then
        EnterLowPowerState();
    else
        if !LocalTimeoutEvent(localtimeout) then
            EnterLowPowerState();
        return;
Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable
// ConstrainUnpredictable()
// ------------------------
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraint ConstrainUnpredictable(Unpredictable which)
  case which of
  when Unpredictable_VMSR
    return Constraint_UNDEF;
  when Unpredictable_WBOVERLAPLD
    return Constraint_WBSUPPRESS; // return loaded value
  when Unpredictable_WBOVERLAPST
    return Constraint_NONE; // store pre-writeback value
  when Unpredictable_LDPOVERLAP
    return Constraint_UNDEF; // instruction is UNDEFINED
  when Unpredictable_BASEOVERLAP
    return Constraint_NONE; // use original address
  when Unpredictable_DATAOVERLAP
    return Constraint_NONE; // store original value
  when Unpredictable_DEVPAGE2
    return Constraint_FAULT; // take an alignment fault
  when Unpredictable_DEVICETAGSTORE
    return Constraint_NONE; // Do not take a fault
  when Unpredictable_INSTRDEVICE
    return Constraint_NONE; // Do not take a fault
  when Unpredictable_RESCPACR
    return Constraint_FALSE; // Map to UNKNOWN value
  when Unpredictable_RESCMAIR
    return Constraint_TRUE; // Map to UNKNOWN value
  when Unpredictable_RESTEXCR
    return Constraint_UNKNOWN; // Map to UNKNOWN value
  when Unpredictable_RESDACR
    return Constraint_UNKNOWN; // Map to UNKNOWN value
  when Unpredictable_RESVTCRS
    return Constraint_UNKNOWN; // Map to UNKNOWN value
  when Unpredictable_RESTnSZ
    return Constraint_FORCE; // Map to the limit value
  when Unpredictable_OORTnSZ
    return Constraint_FORCE; // Map to the limit value
  when Unpredictable_LARGEIPA
    return Constraint_FORCE; // Restrict the input size to the PAMax value
  when Unpredictable_ESRCONDPASS
    return Constraint_FALSE; // Report as "AL"
  when Unpredictable_ILZEROIT
    return Constraint_FALSE; // Do not zero PSTATE.IT
  when Unpredictable_ILZEROT
    return Constraint_FALSE; // Do not zero PSTATE.T
  when Unpredictable_BPVECTORCATCHPRI
    return Constraint_TRUE; // Debug Vector Catch: match on 2nd halfword
  when Unpredictable_VCMATCHHALF
    return Constraint_FALSE; // No match
  when Unpredictable_VCMATCHDAPA
    return Constraint_FALSE; // No match on Data Abort or Prefetch abort
  when Unpredictable_WPMASKANDRAS
    return Constraint_FALSE; // Watchpoint disabled
  when Unpredictable_WPBASCONTIGUOUS
    return Constraint_FALSE; // Watchpoint disabled
  when Unpredictable_WSPWPMASK
    return Constraint_FALSE; // Watchpoint disabled
  when Unpredictable_WPMASKEDBITS
    return Constraint_FALSE; // Watchpoint disabled

when Unpredictable_RESBPMWPCTRL
return Constraint_DISABLED; // Breakpoint/watchpoint disabled
when Unpredictable_BPNOTIMPL
return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_RESBPTYPE
return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPNOTCTXCMP
return Constraint_DISABLED; // Breakpoint disabled
when Unpredictable_BPMATCHHALF
return Constraint_FALSE; // No match
when Unpredictable_BPMISMATCHHALF
return Constraint_FALSE; // Do not force alignment
when Unpredictable_RESTARTALIGNPC
return Constraint_TRUE; // Force zero extension
when Unpredictable_ZERUPOPPER
return Constraint_TRUE; // zero top halves of X registers
when Unpredictable_EFRZERUPOPPERPC
return Constraint_TRUE; // zero top half of PC
when Unpredictable_A32FORCEALIGNPC
return Constraint_FALSE; // Do not force alignment
when Unpredictable_SMD
return Constraint_UNDEF; // disabled SMC is Unallocated
when Unpredictable_NONFAULT
return Constraint_FALSE; // Speculation enabled
when Unpredictable_SVEZEROUPPER
return Constraint_TRUE; // zero top bits of Z registers
when Unpredictable_SVELDNFDATA
return Constraint_TRUE; // Load mem data in NF loads
when Unpredictable_SVELDNFZERO
return Constraint_TRUE; // Write zeros in NF loads
when Unpredictable_CHECKSPNONEACTIVE
return Constraint_TRUE; // Check SP alignment
when Unpredictable_AFUPDATE // AF update for alignment or permission fault
return Constraint_TRUE;
when Unpredictable_SVEZEROUPPERPC
return Constraint_TRUE; // zero top half of PC
when Unpredictable_CLEARERRITEZERO // Clearing sticky errors when instruction in flight
return Constraint_FALSE;
when Unpredictable_ALUEXCEPTIONRETURN
return Constraint_UNDEF;
when Unpredictable_DBGxVR_RESS
return Constraint_FALSE;
when Unpredictable_WFxTDEBUG
return Constraint_FALSE; // WFxT in Debug state does not execute as a NOP
when Unpredictable_LS64UNSUPPORTED
return Constraint_TRUE; // Forces alignment on all instructions when executing 64-bit instructions in little-endian mode
// ConstrainUnpredictableBits()
// ============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that value is always an allocated value; that is, one for which the behavior is not itself CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site. This argument does not appear in the version used in the Armv8 Architecture Reference Manual. See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part of the result, and may not be applicable in all cases.

(Constraint, bits(width)) ConstrainUnpredictableBits(Unpredictable which)

  c = ConstrainUnpredictable(which);
  if c == Constraint_UNKNOWN then
    return (c, Zeros(width));  // See notes; this is an example implementation only
  else
    return (c, bits(width) UNKNOWN);  // bits result not used

// ConstrainUnpredictableBool()
// ============================
// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site. This argument does not appear in the version used in the Armv8 Architecture Reference Manual. See the NOTE on ConstrainUnpredictable() for more information.

boolean ConstrainUnpredictableBool(Unpredictable which)

  c = ConstrainUnpredictable(which);
  assert c IN {Constraint_TRUE, Constraint_FALSE};
  return (c == Constraint_TRUE);

// ConstrainUnpredictableInteger()
// ===============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site. This argument does not appear in the version used in the Armv8 Architecture Reference Manual. See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part of the result.

(Constraint, integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)

  c = ConstrainUnpredictable(which);
  if c == Constraint_UNKNOWN then
    return (c, low);  // See notes; this is an example implementation only
  else
    return (c, integer UNKNOWN);  // integer result not used
```
enum Constraint {
    Constraint_NONE,       // Instruction executes with no change or side-effect to its described behavior
    Constraint_UNKNOWN,    // Destination register has UNKNOWN value
    Constraint_UNDEF,      // Instruction is UNDEFINED
    Constraint_UNDEFEL0,   // Instruction is UNDEFINED at EL0 only
    Constraint_NOP,        // Instruction executes as NOP
    Constraint_TRUE,       // Instruction executes unconditionally
    Constraint_FALSE,      // Instruction executes conditionally
    Constraint_DISABLED,   // Instruction executes with additional decode
    Constraint_ADDITIONAL_DECODE, // Instruction executes unconditionally
    Constraint_WBSUPPRESS, // Instruction executes conditionally
    Constraint_FAULT,      // Instruction executes with additional decode
    Constraint_LIMITED_ATOMICITY, // Accesses are not single-copy atomic above the byte level
    Constraint_FORCE,      // Instruction executes conditionally
    Constraint_FORCENOSLCHECK
};
```
Library pseudocode for shared/functions/unpredictable/Unpredictable
enumeration Unpredictable { // VMSR on MVFR
    Unpredictable_VMSR,
    // Writeback/transfer register overlap (load)
    Unpredictable_WBOVERLAPLD,
    // Writeback/transfer register overlap (store)
    Unpredictable_WBOVERLAPST,
    // Load Pair transfer register overlap
    Unpredictable_LPPOVERLAP,
    // Store-exclusive base/status register overlap
    Unpredictable_BASEOVERLAP,
    // Store-exclusive data/status register overlap
    Unpredictable_DATAOVERLAP,
    // Load-store alignment checks
    Unpredictable_DEVPAGE2,
    // Instruction fetch from Device memory
    Unpredictable_INSTRDEVICE,
    // Reserved CPACR value
    Unpredictable_RESCPACR,
    // Reserved MAIR value
    Unpredictable_RESMAIR,
    // Reserved TEx:C:B value
    Unpredictable_RESTEXCB,
    // Reserved PRRR value
    Unpredictable_RESPRRR,
    // Reserved DACR field
    Unpredictable_RESDACR,
    // Reserved TCR.TnSZ value
    Unpredictable_RESVTcrs,
    // Reserved TCR.TnSZ value
    Unpredictable_RESTnSZ,
    // Tag stored to Device memory
    Unpredictable_DEVICETAGSTORE,
    // Reserved SCTRL_ELx.EnCSR value
    Unpredictable_RESEnCSR,
    // Reserved CSRcR_ELx.SIZE value
    Unpredictable_RESCSRCRSIZE,
    // Out-of-range TCR.TnSZ value
    Unpredictable_OORTnSZ,
    // IPA size exceeds PA size
    Unpredictable_LARGEIPA,
    // Syndrome for a known-passing conditional A32 instruction
    Unpredictable_ESRCONDPASS,
    // Illegal State exception: zero PSTATE.IT
    Unpredictable_ILZEROIT,
    // Illegal State exception: zero PSTATE.T
    Unpredictable_ILZEROT,
    // Debug: prioritization of Vector Catch
    Unpredictable_BPVECTOCATCHPRI,
    // Debug Vector Catch: match on 2nd halfword
    Unpredictable_VCMATCHHALF,
    // Debug Vector Catch: match on Data Abort or Prefetch abort
    Unpredictable_VCMATCHDAPA,
    // Debug watchpoints: non-zero MASK and non-ones BAS
    Unpredictable_WPBMASKANDDBAS,
    // Debug watchpoints: non-contiguous BAS
    Unpredictable_WPBASENCOUNTINUOUS,
    // Debug watchpoints: reserved MASK
    Unpredictable_RESWPBMASK,
    // Debug watchpoints: non-zero MASKed bits of address
    Unpredictable_WPBMASKEDBITS,
    // Debug breakpoints and watchpoints: reserved control bits
    Unpredictable_RESBPWPCTRL,
    // Debug breakpoints: not implemented
    Unpredictable_BPNOTIMPL,
    // Debug breakpoints: reserved type
    Unpredictable_RESBPTYPE,
    // Debug breakpoints: not-context-aware breakpoint
    Unpredictable_BPNOTCTXCMP,
// Debug breakpoints: match on 2nd halfword of instruction
Unpredictable_BPMATCHHALF,
// Debug breakpoints: mismatch on 2nd halfword of instruction
Unpredictable_BPMISMATCHHALF,
// Debug: restart to a misaligned AArch32 PC value
Unpredictable_RESTARTALIGNPC,
// Debug: restart to a not-zero-extended AArch32 PC value
Unpredictable_RESTARTZEROUPPERPC,
// Zero top 32 bits of X registers in AArch32 state
Unpredictable_ZEROUPPER,
// Zero top 32 bits of PC on illegal return to AArch32 state
Unpredictable_ERETZEROUPPERPC,
// Force address to be aligned when interworking branch to A32 state
Unpredictable_A32FORCEALIGNPC,
// SMC disabled
Unpredictable_SMD,
// FF speculation
Unpredictable_NONFAULT,
// Zero top bits of Z registers in EL change
Unpredictable_SVEZEROUPPER,
// Load mem data in NF loads
Unpredictable_SVELDNFDATA,
// Write zeros in NF loads
Unpredictable_SVELDNFZERO,
// SP alignment fault when predicate is all zero
Unpredictable_CHECKSPNONEACTIVE,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Consider SCTLR[].IESB in Debug state
Unpredictable_IESBinDebug,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,
// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
// WFET or WFIT instruction in Debug state
Unpredictable_WFxTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO,
// ALUEXCEPTIONRETURN when in user/system mode in A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Compare DBGVR.RESS for BP/WP
Unpredictable_DBGxVR_RESS};
Library pseudocode for shared/functions/vector/AdvSIMDEndExpandImm

// AdvSIMDEndExpandImm()
// ================

bits(64) AdvSIMDEndExpandImm(bit op, bits(4) cmode, bits(8) imm8)
  case cmode<3:1> of
    when '000'
      imm64 = Replicate(Zeros(24):imm8, 2);
    when '001'
      imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
    when '010'
      imm64 = Replicate(imm8:Zeros(24), 2);
    when '011'
      if cmode<0> == '0' then
        imm64 = Replicate(Zeros(8):imm8:Ones(8), 4);
      else
        imm64 = Replicate(imm8:Zeros(8), 4);
    when '100'
      imm64 = Replicate(imm8:Zeros(8), 4);
    when '101'
      if cmode<0> == '0' then
        imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
      else
        imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
    when '111'
      if cmode<0> == '0' & op == '0' then
        imm64 = Replicate(imm8, 8);
      if cmode<0> == '0' & op == '1' then
        imm64 = Replicate(imm8<7>, 8); imm64b = Replicate(imm8<6>, 8);
        imm6c = Replicate(imm8<5>, 8); imm6d = Replicate(imm8<4>, 8);
        imm6e = Replicate(imm8<3>, 8); imm6f = Replicate(imm8<2>, 8);
        imm6g = Replicate(imm8<1>, 8); imm6h = Replicate(imm8<0>, 8);
      if cmode<0> == '1' & op == '0' then
        imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<5>:imm8<4>:imm8<3>:imm8<2>:imm8<1>:imm8<0>, 8);
        imm64 = Replicate(imm32, 2);
      if cmode<0> == '1' & op == '1' then
        if UsingAArch32() then ReservedEncoding();
        imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<5>:imm8<4>:imm8<3>:imm8<2>:imm8<1>:imm8<0>, 8);
      imm64a = Replicate(imm8<7>, 8); imm68b = Replicate(imm8<6>, 8);
      imm6c = Replicate(imm8<5>, 8); imm6d = Replicate(imm8<4>, 8);
      imm6e = Replicate(imm8<3>, 8); imm6f = Replicate(imm8<2>, 8);
      imm6g = Replicate(imm8<1>, 8); imm6h = Replicate(imm8<0>, 8);
    return imm64;

Library pseudocode for shared/functions/vector/MatMulAdd

// MatMulAdd()
// ===========

// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
  assert N == 128;
  bits(N) result;
  bits(32) sum;
  integer prod;

  for i = 0 to 1
    for j = 0 to 1
      sum = Elem[addend, 2*i+j, 32];
      for k = 0 to 7
        prod = Int(Elem[op1, 8*i+k, 8], op1_unsigned) * Int(Elem[op2, 8*j+k, 8], op2_unsigned);
        sum = sum + prod;
      Elem[result, 2*i+j, 32] = sum;
  return result;
Library pseudocode for shared/functions/vector/PolynomialMult

// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;

Library pseudocode for shared/functions/vector/SatQ

// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);

Library pseudocode for shared/functions/vector/SignedSatQ

// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<\text{N-1:0}>, saturated);

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    if operand<\text{N-1:N-2}> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        estimate = RecipSqrtEstimate(UInt(operand<31:23>), FALSE);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);
    return result;
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
assert N == 32;
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)
        estimate = RecipEstimate(UInt(operand<31:23>), FALSE);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

return result;

// UnsignedSatQ()
// ===============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elseif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;

return (result<N-1:0>, saturated);

// SelfHostedTraceEnabled()
// ========================

// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    if !HaveTraceExt() || !HaveSelfHostedTrace() then return FALSE;
    if HaveEL(EL3) then
        secure_trace_enable = (if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE);
        niden = (secure_trace_enable == '0' || ExternalSecureNoninvasiveDebugEnabled());
    else
        // If no EL3, IsSecure() returns the Effective value of (SCR_EL3.NS == '0')
        niden = (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled());

return (EDSCR.TFO == '0' || !niden);
Library pseudocode for shared/trace/selfhosted/TraceAllowed

// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the current Security state and Exception Level
boolean TraceAllowed()
    if !HaveTraceExt() then return FALSE;
    if SelfHostedTraceEnabled() then
        if IsSecure() && HaveEL(EL3) then
            secure_trace_enable = (if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE);
            if secure_trace_enable == '0' then return FALSE;
            TGE_bit = if EL2Enabled() then HCR_EL2.TGE else '0';
            case PSTATE.EL of
                when EL3 TRE_bit = if HighestELUsingAArch32() then TRFCR.E1TRE else '0';
                when EL2 TRE_bit = TRFCR_EL2.E2TRE;
                when EL1 TRE_bit = TRFCR_EL1.E1TRE;
                when EL0 TRE_bit = if TGE_bit == '1' then TRFCR_EL2.E0HTRE else TRFCR_EL1.E0TRE;
                    return TRE_bit == '1';
            else
                return (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled());
    else
        return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');

Library pseudocode for shared/trace/selfhosted/TraceContextIDR2

// TraceContextIDR2()
// ================

boolean TraceContextIDR2()
    if !TraceAllowed() || !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');

Library pseudocode for shared/trace/selfhosted/TraceSynchronizationBarrier

// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers
TraceSynchronizationBarrier();
Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            TS_el2 = TRFCR_EL2.TS;
            if TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP);
            case TS_el2 of
                when '00' // Falls through to check TRFCR_EL1.TS
                when '01'
                    return TimeStamp_Virtual;
                when '10'
                    assert HaveECVExt();
                    return TimeStamp_OffsetPhysical;
                when '11'
                    return TimeStamp_Physical;
                otherwise
                    Unreachable(); // ConstrainUnpredictableBits removes this case
            end case
        TS_el1 = TRFCR_EL1.TS;
        if TS_el1 == 'x0' then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP);
        case TS_el1 of
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt();
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable(); // ConstrainUnpredictableBits removes this case
        else
            return TimeStamp_CoreSight;
        end case
    end if
end if

Library pseudocode for shared/trace/system/IsTraceCorePowered

// Returns TRUE if the Trace Core Power Domain is powered up
boolean IsTraceCorePowered();
Library pseudocode for shared/translation/attrs/CombineS1S2AttrHints

// CombineS1S2AttrHints()
// ======================
// Combines cacheability attributes and allocation hints from stage 1 and stage 2

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc, AccType s2acctype)

MemAttrHints result;

apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
if apply_force_writeback then
  if S2CacheDisabled(s2acctype) then
    result.attrs = MemAttr_NC;    // force Non-cacheable
  elsif s2desc.attrs == '11' then
    result.attrs = s1desc.attrs;
  elsif s2desc.attrs == '10' then
    result.attrs = MemAttr_WB;    // force Write-back
  else
    result.attrs = MemAttr_NC;
  end
else
  if s2desc.attrs == '01' || s1desc.attrs == '01' then
    result.attrs = bits(2) UNKNOWN;   // Reserved
  elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
    result.attrs = MemAttr_NC;        // Non-cacheable
  elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
    result.attrs = MemAttr_WT;        // Write-through
  else
    result.attrs = MemAttr_WB;        // Write-back
  end
if result.attrs == MemAttr_NC then
  result.hints = MemHint_No;
elsif apply_force_writeback then
  if s1desc.attrs != MemAttr_NC then
    result.hints = s1desc.hints;
  else
    result.hints = MemHint_RWA;
  end
else
  result.hints = s1desc.hints;
result.transient = s1desc.transient;
return result;

Library pseudocode for shared/translation/attrs/CombineS1S2Device

// CombineS1S2Device()
// ===================
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
  result = DeviceType_nGnRnE;
elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
  result = DeviceType_nGnRE;
elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
  result = DeviceType_nGRE;
else
  result = DeviceType_GRE;
return result;
Library pseudocode for shared/translation/attrs/LongConvertAttrsHints

// LongConvertAttrsHints()
// =======================
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);
    MemAttrHints result;
    if SICacheDisabled(acctype) then // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elsif attrfield<3:0> == '0100' then // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elsif attrfield<3:2> == '01' then // Write-back transient
            result.attrs = MemAttr_WB;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        else // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;
    return result;

Library pseudocode for shared/translation/attrs/MemAttrDefaults

// MemAttrDefaults()
// =================
// Supply default values for memory attributes, including overriding the shareability attributes
// for Device and Non-cacheable memory types.

MemoryAttributes MemAttrDefaults(MemoryAttributes memattrs)
    if memattrs.memtype == MemType_Device then
        memattrs.inner = MemAttrHints UNKNOWN;
        memattrs.outer = MemAttrHints UNKNOWN;
        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;
    else
        memattrs.device = DeviceType UNKNOWN;
        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
            memattrs.shareable = TRUE;
            memattrs.outershareable = TRUE;
    return memattrs;
Library pseudocode for shared/translation/attrs/S1CacheDisabled

// S1CacheDisabled()
// ================

boolean S1CacheDisabled(AccType acctype)
if ELUsingAArch32(S1TranslationRegime()) then
  if PSTATE.EL == EL2 then
    enable = if acctype == AccType_IFETCH then HSCTRL.I else HSCTRL.C;
  else
    enable = if acctype == AccType_IFETCH then SCTLR.I else SCTLR[].C;
else
  enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
return enable == '0';

Library pseudocode for shared/translation/attrs/S2AttrDecode

// S2AttrDecode()
// ==============
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)
MemoryAttributes memattrs;
apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';
// Device memory
if (apply_force_writeback && attr<2> == '0') || attr<3:2> == '00' then
  memattrs.memtype = MemType_Device;
  case attr<1:0> of
    when '00'  memattrs.device = DeviceType_nGnRnE;
    when '01'  memattrs.device = DeviceType_nGnRE;
    when '10'  memattrs.device = DeviceType_nGRE;
    when '11'  memattrs.device = DeviceType_GRE;
// Normal memory
elsif apply_force_writeback then
  if attr<2> == '1' then
    memattrs.memtype = MemType_Normal;
    memattrs.inner.attrs = attr<1:0>;
    memattrs.outer.attrs = attr<1:0>;
    memattrs.shareable = SH<1> == '1';
    memattrsoutershareable = SH == '10';
  else
    memattrs = MemoryAttributes UNKNOWN; // Reserved
  return MemAttrDefaults(memattrs);

else
  if attr<1:0> != '00' then
    memattrs.memtype = MemType_Normal;
    memattrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
    memattrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
    memattrs.shareable = SH<1> == '1';
    memattrsoutershareable = SH == '10';
  else
    memattrs = MemoryAttributes UNKNOWN; // Reserved
  return MemAttrDefaults(memattrs);

Library pseudocode for shared/translation/attrs/S2CacheDisabled

// S2CacheDisabled()
// ================

boolean S2CacheDisabled(AccType acctype)
if ELUsingAArch32(EL2) then
  disable = if acctype == AccType_IFETCH then HCR2.ID else HCR2.CD;
else
  disable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
return disable == '1';
// S2ConvertAttrsHints()
// =====================
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
assert attr != '00';
MemAttrHints result;
if S2CacheDisabled(acctype) then                // Force Non-cacheable
    result.attrs = MemAttr_NC;
    result.hints = MemHint_No;
else
    case attr of
        when '01'                               // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
        when '10'                               // Write-through
            result.attrs = MemAttr_WT;
            result.hints = MemHint_RWA;
        when '11'                               // Write-back
            result.attrs = MemAttr_WB;
            result.hints = MemHint_RWA;
    end case;
result.transient = FALSE;
return result;

// ShortConvertAttrsHints()
// ========================
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype, boolean secondstage)
MemAttrHints result;
if (!secondstage && S1CacheDisabled(acctype)) || (secondstage && S2CacheDisabled(acctype)) then
    // Force Non-cacheable
    result.attrs = MemAttr_NC;
    result.hints = MemHint_No;
else
    case RGN of
        when '00'                   // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
        when '01'                   // Write-back, Read and Write allocate
            result.attrs = MemAttr_WB;
            result.hints = MemHint_RWA;
        when '10'                   // Write-through, Read allocate
            result.attrs = MemAttr_WT;
            result.hints = MemHint_RA;
        when '11'                   // Write-back, Read allocate
            result.attrs = MemAttr_WB;
            result.hints = MemHint_RA;
    end case;
result.transient = FALSE;
return result;
Library pseudocode for shared/translation/attrs/WalkAttrDecode

// WalkAttrDecode()
// ================
MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)

    MemoryAttributes memattrs;
    AccType acctype = AccType_NORMAL;
    memattrs.memtype = MemType_Normal;
    memattrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
    memattrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
    memattrs.shareable = SH<1> == '1';
    memattrs.outershareable = SH == '10';
    memattrs.tagged = FALSE;
    return MemAttrDefaults(memattrs);

Library pseudocode for shared/translation/translation/HasS2Translation

// HasS2Translation()
// ===============
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
    return (EL2Enabled() && !IsInHost() && PSTATE.EL IN {EL0, EL1});

Library pseudocode for shared/translation/translation/Have16bitVMID

// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID";

Library pseudocode for shared/translation/translation/PAMax

// PAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
Library pseudocode for shared/translation/translation/S1TranslationRegime

// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;
end;

Library pseudocode for shared/translation/translation/VAMax

// VAMax()
// ========
// Returns the IMPLEMENTATION DEFINED upper limit on the virtual address
// size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";

Internal version only: isa v32.12, AdvSIMD v29.04, pseudocode v2020-09_xml, sve v2020-09_rc2b ; Build timestamp: 2020-09-30T20:20
Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.