Arm® A64 Instruction Set Architecture,
for Arm A-class

Future Architecture Technologies in the A architecture profile

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

Copyright © 2018-2019 Arm Limited (or its affiliates). All rights reserved.

Release Information

For information on the change history and known issues for this release, see the Release notes in the A64 XML for Future Architecture Technologies (201909)

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the Arm’s trademark usage guidelines http://www.arm.com/company/policies/trademarks.

Copyright © 2018-2019 Arm Limited (or its affiliates). All rights reserved.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is for a Beta product, that is a product under development.

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.

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

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

**ADDX (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).

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

**ANSD (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.

**BFM**: Arithmetic Shift Right Variable.

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

**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.
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.

**LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB**: Atomic bit clear on byte in memory.

**LDCLRH, LDCLRAH, 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, LDSETAL, LDSETL: Atomic bit set on word or doubleword in memory.
LDSETB, LDSETAB, LDSETALB, LDSETLB: Atomic bit set on byte in memory.
LDSETH, LDSETAH, LDSETALH, LDSETLH: 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, LDMINI: 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, LDUAXL: 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 (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: Negate: an alias of SUBS (shifted register).

NGC: Negate with Carry: an alias of SBC.

NGS: 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, PACDZB: 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.

**STADD, STADDL**: Atomic add on word or doubleword in memory, without return: an alias of LDADD, LDADDA, LDADDAL, LDADDL.

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

**STADDDH, STADDLH**: Atomic add on halfword in memory, without return: an alias of LDADDDH, 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.

**STXHR**: 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).

**SUBLG**: 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.

SXTB: 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).


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.

**WFI**: Wait For Interrupt.

**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 0 0 0</th>
<th>Rm</th>
<th>0 0 0 0 0 0 0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op 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>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>
</tr>
</tbody>
</table>

32-bit ($sf == 0$)

$$ADCS \langle Wd \rangle, \langle Wn \rangle, \langle Wm \rangle$$

64-bit ($sf == 1$)

$$ADCS \langle Xd \rangle, \langle Xn \rangle, \langle Xm \rangle$$

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
integer $m = \text{UInt}(Rm)$;
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;

$\langle result, nzcv \rangle = \text{AddWithCarry}(\text{operand1}, \text{operand2}, \text{PSTATE.C})$;

PSTATE.$<N,Z,C,V>$ = nzcv;

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

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.
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.

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\hline
sf & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & Rm & option & imm3 & Rn & Rd & \\
\hline
op & S & \\
\end{array}
\]

32-bit (sf == 0)

\[
ADD <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> \{<amount>\}}
\]

64-bit (sf == 1)

\[
ADD <Xd|SP>, <Xn|SP>, <R><m>{, <extend> \{<amount>\}}
\]

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

\(<\text{extend}>\) 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;\text{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 "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 \(<\text{extend}>\) 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);
(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>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 0 0 0 0 1 0</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>}

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

```plaintext
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 == '00000000000' &amp;&amp; (Rd == '11111'</td>
</tr>
</tbody>
</table>
Operation

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

(result, -) = AddWithCarry(operand1, imm, '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 (shifted register)

Add (shifted register) adds 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>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift</td>
<td>0</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>
</tr>
<tr>
<td>imm6</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>
</tr>
<tr>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

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

```
ADD <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.

### Operation

```
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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | uimm6 | [0](0) | uimm4 | Xn | Xd |

**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);

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() then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';
(result, -) = AddWithCarry(operand1, offset, '0');
result = AArch64.AddressWithAllocationTag(result, rtag);
if d == 31 then
    SP[] = result;
else
    X[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09 rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ADD (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)**.

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

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

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

ADD <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”:

<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'.
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 == '1111'</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 \texttt{CMN (immediate)}.

<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>
<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>
</tr>
</tbody>
</table>

32-bit (\texttt{sf == 0})

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

64-bit (\texttt{sf == 1})

\texttt{ADDS <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;

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

Assembler Symbols

- \texttt{<Wd>} is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.
- \texttt{<Wn|WSP>} is the 32-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.
- \texttt{<Xd>} is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.
- \texttt{<Xn|SP>} is the 64-bit name of the source general-purpose register or stack pointer, encoded in the “Rn” field.
- \texttt{<imm>} is an unsigned immediate, in the range 0 to 4095, encoded in the “imm12” field.
- \texttt{<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 == '11111'</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).

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

32-bit (sf == 0)

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

64-bit (sf == 1)

ADDS <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 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 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| immlo | 1 | 0 | 0 | 0 | 0 | immhi |
```

\text{op}

**Literal**

\text{ADR \langle Xd \rangle, \langle label \rangle}

\text{integer d = UInt(Rd);} \\
\text{bits(64) imm;}

\text{imm = SignExtend(immhi:immlo, 64);}

**Assembler Symbols**

\text{\langle Xd \rangle} \quad \text{Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.}

\text{\langle label \rangle} \quad \text{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**

\text{bits(64) base = PC[];} \\
\text{X[d] = base + imm;}

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

```

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 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   | immlo| 1   | 0   | 0   | 0   | 0   | immhi|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
```

**Literal**

```
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);
Xd[d] = base + imm;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

$$\begin{array}{cccccccccccccc}
\text{sf} & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & N & \text{immr} & \text{imms} & \text{Rn} & \text{Rd} & \text{opc}
\end{array}$$

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

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

64-bit ($sf == 1$)

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

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
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.
**AND (shifted register)**

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.

<table>
<thead>
<tr>
<th>sf</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>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>N</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>}
```

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

```
bits(datasize) operand1 = X[n];
bites(datasize) 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).

```
31 30 29 28 27 26 25 24 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 0 1 0 0 N   immr |   immr |   immr |   immr |   immr |   immr |   immr |   immr |   immr |   immr |   immr |   immr | Rd
```

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

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

64-bit (sf == 1)

ANDS <Xd>, <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>`: 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>TST (immediate)</td>
<td>Rd == '11111'</td>
</tr>
</tbody>
</table>

Operation

```
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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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)**.

<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>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>
</tbody>
</table>

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

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

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

ANDS <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.

**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>
Operation

\[
\begin{align*}
\text{bits(\text{datasize}) operand1} &= X[n]; \\
\text{bits(\text{datasize}) operand2} &= \text{ShiftReg}(m, \text{shift\_type, shift\_amount}); \\
\text{result} &= \text{operand1 AND operand2;} \\
\text{PSTATE.<N,Z,C,V>} &= \text{result<\text{datasize}-1>:IsZeroBit(result)':'00'}; \\
X[d] &= \text{result};
\end{align*}
\]

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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

ASRV \(<Wd>, <Wn>, <Wm>\)

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

ASRV \(<Xd>, <Xn>, <Xm>\)

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);
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.
**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 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  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 1  | 1  | 0  |    |    |    |    |    |    |    |    |    |
```

**AUTDA (Z == 0)**

AUTDA <Xd>, <Xn|SP>

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

AUTDZA <Xd>

```java
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**

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

---

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_r2_c1, sve v2019-09_r3; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 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.

**Integer**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | Z | 1 | 1 | 1 | Rn | Rd |

**AUTDB (Z == 0)**

AUTDB <Xd>, <Xn|SP>

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

AUTDZB <Xd>

```plaintext
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**

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

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | Z | 1  | 0  | 0  |

**AUTIA** (Z == 0)

AUTIA <Xd>, <Xn|SP>

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

AUTIZA <Xd>

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  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | x  | 1  | 1  | 0  | x  | 1  | 1  | 1  | 1  | 1  |
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
        X[d] = AuthIA(X[d], SP[], FALSE);
    else
        X[d] = AuthIA(X[d], X[n], FALSE);

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 1  | 0  | 0  | 0  | 0  | 1  | 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  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | x | 1  | 1  | 1  | x | 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
    \texttt{X[d]} = \texttt{AuthIB(X[d], SP[], FALSE)};
  else
    \texttt{X[d]} = \texttt{AuthIB(X[d], X[n], FALSE)};

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)

if !HaveFlagFormatExt() then UNDEFINED;

Operation

bit Z = PSTATE.Z OR PSTATE.V;
bit 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | imm19 | 0  | cond |

**19-bit signed PC-relative branch offset**

B.<cond> <label>

bits(64) offset = \texttt{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**

\[
\text{if } \text{ConditionHolds}(\text{cond}) \text{ then} \\
\quad \text{BranchTo}(\text{PC}[] + \text{offset}, \text{BranchType_DIR});
\]

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

26-bit signed PC-relative branch offset

B <label>

bits(64) offset = \text{SignExtend}(\text{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

\text{BranchTo}(\text{PC[]} + \text{offset}, \text{BranchType_DIR});
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 other bits of the destination register remain unchanged.

This instruction is used by the aliases \texttt{BFC}, \texttt{BFI}, and \texttt{BFXIL}.

### Assembler Symbols

- \(<\text{Wd}>\) Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{Wn}>\) Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
- \(<\text{Xd}>\) Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
- \(<\text{Xn}>\) 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>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{BFC}</td>
<td>( \text{Rn} == '11111' ) &amp;&amp; \texttt{UInt}(imms) &lt; \texttt{UInt}(immr)</td>
</tr>
<tr>
<td>\texttt{BFI}</td>
<td>( \text{Rn} != '11111' ) &amp;&amp; \texttt{UInt}(imms) &lt; \texttt{UInt}(immr)</td>
</tr>
<tr>
<td>\texttt{BFXIL}</td>
<td>\texttt{UInt}(imms) &gt;= \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.
**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>Bit</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>sf</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**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>ShiftType</th>
<th>Description</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>

**Operation**

```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;
integer shift_amount = UInt(imm6);

ShiftType shift_type = DecodeShift(shift);

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;
integer shift_amount = UInt(imm6);

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

```plaintext
int sf = 0;
int shift = 1;
int Rm = 0;
int imm6 = 0;
int Rn = 0;
int Rd = 0;
```

```
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;
integer shift_amount = UInt(imm6);

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = 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.
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>sf</th>
<th>1 1 0 1 0 1 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>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 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

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.
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.

\[
\begin{array}{cccccccccccccccccc}
\hline
1 & 0 & 0 & 1 & 0 & 1 \hline
\end{array}
\]

**imm26**

**26-bit signed PC-relative branch offset**

```
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**

```
X[30] = PC[] + 4;

BranchTo(PC[] + offset, BranchType_DIRCALL);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

<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 0 1 1 0 1 1 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 Rn 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**Integer**

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];

\[ X[30] = PC[] + 4; \]

BranchTo(target, BranchType_INDCALL);
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  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);

X[30] = PC[] + 4;

BranchTo(target, BranchType_INDCALL);
Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

Assembler Symbols

\[
\text{BR } \langle \text{Xn} \rangle
\]

integer \( n = \text{UInt}(Rn); \)

Operation

\[
\text{bits(64) target} = X[n];
\]

\[
\text{BranchTo(target, BranchType_INDIR)};
\]
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)

```
<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>Z</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>M</td>
<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>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
```

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

```plaintext
BRAAZ <Xn>
```

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

```plaintext
BRAA <Xn>, <Xm|SP>
```

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

```plaintext
BRABZ <Xn>
```

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

```plaintext
BRAB <Xn>, <Xm|SP>
```

```
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);

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.

```
| 31 30 29 28 27 26 25 24 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 | imm16 | 0 0 0 0 0 |
```

System

```
BRK #<imm>
- = HaveBTIExt();
```

Assembler Symbols

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

Operation

```
AArch64.SoftwareBreakpoint(imm16);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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)*

```
System

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

**BTI**

```
SystemHintOp op;
if CRm:op2 == '0100 xx0' then
  op = SystemHintOp_BTIBT;
  // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
  SetBTypeCompatible(BTypeCompatible_BTIBT(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
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, TRUE);
      if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, TRUE);
      WaitForEvent();
  when SystemHintOp_WFI
    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();
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    if 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
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 no memory ordering requirements.

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   | 0   | 0   | 0   | 0   | 0   | 1   | L   | L   | Rs  | o0  | 1   | 1   | 1   | 1   | 1   | 1   | 1   | Rn  |     |     |     |     |     |     |     |     |

size
32-bit CAS (size == 10 & L == 0 & o0 == 0)
  CAS <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASA (size == 10 & L == 1 & o0 == 0)
  CASA <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASAL (size == 10 & L == 1 & o0 == 1)
  CASAL <Ws>, <Wt>, [<Xn|SP>{,#0}]

32-bit CASL (size == 10 & L == 0 & o0 == 1)
  CASL <Ws>, <Wt>, [<Xn|SP>{,#0}]

64-bit CAS (size == 11 & L == 0 & o0 == 0)
  CAS <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASA (size == 11 & L == 1 & o0 == 0)
  CASA <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASAL (size == 11 & L == 1 & o0 == 1)
  CASAL <Xs>, <Xt>, [<Xn|SP>{,#0}]

64-bit CASL (size == 11 & L == 0 & o0 == 1)
  CASL <Xs>, <Xt>, [<Xn|SP>{,#0}]

if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
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.
<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
    SetNotTagCheckedInstruction(!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.

- CASB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has no memory ordering requirements.

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)**

<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 1</td>
</tr>
</tbody>
</table>

| size |

CASAB (L == 1 && o0 == 0)

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

CASALB (L == 1 && o0 == 1)

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

CASB (L == 0 && o0 == 0)

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

CASLB (L == 0 && o0 == 1)

CASLB <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_ORDEREDATOMIRCW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMIRCW 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(8) comparevalue;
bits(8) newvalue;
bits(8) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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 0 0 1 1 | L | 1 | Rs | o0 | 1 1 1 1 | Rn | Rt |

**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
    SetNotTagCheckedInstruction(!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);
CAS, 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 no memory ordering requirements.

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)

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

Rt2
32-bit CASP (sz == 0 & L == 0 & o0 == 0)

CASP <Ws>, <W(s+1)>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPA (sz == 0 & L == 1 & o0 == 0)

CASPA <Ws>, <W(s+1)>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPAL (sz == 0 & L == 1 & o0 == 1)

CASPAL <Ws>, <W(s+1)>, <W(t+1)>, [<Xn|SP>{,#0}]

32-bit CASPL (sz == 0 & L == 0 & o0 == 1)

CASPL <Ws>, <W(s+1)>, <W(t+1)>, [<Xn|SP>{,#0}]

64-bit CASP (sz == 1 & L == 0 & o0 == 0)

CASP <Xs>, <X(s+1)>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPA (sz == 1 & L == 1 & o0 == 0)

CASPA <Xs>, <X(s+1)>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPAL (sz == 1 & L == 1 & o0 == 1)

CASPAL <Xs>, <X(s+1)>, <X(t+1)>, [<Xn|SP>{,#0}]

64-bit CASPL (sz == 1 & L == 0 & o0 == 1)

CASPL <Xs>, <X(s+1)>, <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.
<X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.
<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(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() then s1:s2 else s2:s1;
newvalue = if BigEndian() then t1:t2 else t2:t1;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);
if n == 31 then
  CheckSPAlignment();
  address = SP[];
else
  address = X[n];

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

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

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

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
| sf | 0 | 1 | 1 | 0 | 0 | 1 | | imm19 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Rt |

op

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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.

<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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>imm19</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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);

---

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CCMN (immediate)

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.

```
<table>
<thead>
<tr>
<th>sf</th>
<th>Rm</th>
<th>cond</th>
<th>Rn</th>
<th>nzcv</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**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**

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

```
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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

### 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>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>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>
<td></td>
<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>
<-op-

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

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

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

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

```plaintext
gerint n = UInt(Rn);
gerint m = UInt(Rm);
gerint 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**

```plaintext
bits(datasize) operand1 = X[n];
b 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     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) |
```

CRm

System

```
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.
CLREX

Clear Exclusive clears the local monitor of the executing PE.

\[
\begin{array}{cccccccccccccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & CRm & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

System

\[
\text{CLREX} \{\#<\text{imm}>\}
\]

// 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

\[
\text{ClearExclusiveLocal}(\text{ProcessorID}());
\]
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.

32-bit (sf == 0)

CLS <Wd>, <Wn>

64-bit (sf == 1)

CLS <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 = 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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 | 1 | 0 | 0 | 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 |

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.
**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.

### Assembly 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.

<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>sf 0 0 1 1 0 1 0 1 1 0 Rm 0 1 0 1 sz Rn Rd</td>
</tr>
</tbody>
</table>

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

bits(32) acc = X[n];  // accumulator
bits(size) val = X[m];  // input value
bits(32) poly = 0x1EDC6F41<31:0>;

bits(size+32) 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.
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.

```
31 30 29 28 27 26 25 24 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 0 1 0 1 0 0 1 1 1 1 1
```

System

CSDB

// Empty.

Operation

```
ConsumptionOfSpeculativeDataBarrier();
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
CSEL

Conditional Select 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.

31 30 29 28 27 26 25 24 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 0 0 Rm cond 0 0 Rn Rd
op o2

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.
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

Alias Is preferred when
--- ---
CINC Rx != '1111' && cond != '111x' && Rx != '1111' && Rx == Rx
CSET Rx == '1111' && cond != '111x' && Rx == '11111'

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.

|   |   | sf |   |   | op | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|   |   | Rd |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINV</td>
<td>Rm != '11111' &amp;&amp; cond != '111x' &amp;&amp; Rn != '11111' &amp;&amp; Rn == Rm</td>
</tr>
<tr>
<td>CSETM</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 = 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.
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 0 1 0 1 0 1 0</th>
<th>Rm</th>
<th>cond</th>
<th>0 1</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>02</td>
<td></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>

```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.

<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;&amp; Rn == Rm</td>
</tr>
</tbody>
</table>

#### Operation

```plaintext
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.
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:
• \textit{ELR}_\textit{ELx} becomes \textit{UNKNOWN}.
• \textit{SPSR}_\textit{ELx} becomes \textit{UNKNOWN}.
• \textit{ESR}_\textit{ELx} becomes \textit{UNKNOWN}.
• \textit{DLR}_\textit{EL0} and \textit{DSPSR}_\textit{EL0} become \textit{UNKNOWN}.
• The endianness is set according to \textit{SCTLR}_\textit{ELx} EE.
This instruction is \texttt{UNDEFINED} at EL0 in Non-secure state if EL2 is implemented and \textit{HCR}_\textit{EL2}.TGE == 1.
This instruction is always \texttt{UNDEFINED} in Non-debug state.
For more information on the operation of the DCPSn instructions, see \textit{DCPS}.

\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
 1 1 0 1 0 1 0 0 | 1 0 1 | imm16 | 0 0 0 | 0 1 | LL
\end{verbatim}

System

\textbf{DCPS1 \{#<imm>\}}

if !Halted() then UNDEFINED;

Assembler Symbols

\textless imm\textgreater 
Is an optional 16-bit unsigned immediate, in the range 0 to 65535, defaulting to 0 and encoded in the "imm16" field.

Operation

\textbf{DCPSInstruction}(LL);

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 $SCTL\_ELx$ 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.

```
31 30 29 28 27 26 25 24 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 | imm16 | 0 0 0 1 0
LL
```

System

```
DCPS2 {#<imm>}

if !Halted() then UNDEFINED;
```

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

System
(Armv8.0-A)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | imm16 | 0  | 0  | 0  | 1  | 1  | |

System

DCPS3 {#<imm>} {#<imm>}

if !Halted() then UNDEFINED;

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);

Copyright © 2010-2019 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)

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  |
```

**Operation**

```
if !HaveDGHExt() then EndOfInstruction();

Hint_DGH();
```
Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see Data Memory Barrier.

System

DMB <option>|#<imm>

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;

case CRm<1:0> of
  when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
  when '01' types = MBReqTypes_Reads;
  when '10' types = MBReqTypesWrites;
  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.

System

DRPS

if !Halted() || PSTATE.EL == EL0 then UNDEFINED;

Operation

DRPSInstruction();
Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see Data Synchronization Barrier.

```
<table>
<thead>
<tr>
<th>CRm</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

System

```
DSB <option>|#<imm>
```

```text
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;

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, 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.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the “CRm” field.

Operation

if TSTATE.depth > 0 then
   FailTransaction(TMFailure_ERR, FALSE);
   DataSynchronizationBarrier(domain, types);
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.

```
<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>shift</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>
</tr>
</tbody>
</table>
```

32-bit (sf == 0)

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

64-bit (sf == 1)

EON <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>` 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 EOR 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.
**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.

```
   31 30 29 28 27 26 25 24 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  0  0  N    immr  |  |  |  |  |  |  |  |  imms  |  |  |  |  |  |  |  |  Rd  |  |  |  |  |  |  |  |  Rn  |
```

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

EOR `<Wd|WSP>`, `<Wn>`, `#<imm>`

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

EOR `<Xd|SP>`, `<Xn>`, `#<imm>`

integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bids(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;
bids(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>1 0</th>
<th>0</th>
<th>1 0</th>
<th>1 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>N</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

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

\[
\text{bits(datasize)} \text{ operand2} = \text{ShiftReg}(m, \text{shift_type}, \text{shift_amount});
\]

\[
\text{result} = \text{operand1} \text{ EOR operand2};
\]

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

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.
ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores `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 `Illegal return events from AArch64 state`.

ERET is UNDEFINED at EL0.

```
1 1 0 1 0 1 1 0 1 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
A M Rn op4
```

System

```
ERET

if PSTATE.EL == EL0 then UNDEFINED;
```

Operation

```
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR[];
AArch64.ExceptionReturn(target, SPSR[]);
```
ERETAA, ERETAB

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 | 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  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | M | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |   |
| A  | Rn | op4|

**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)

```
31 30 29 28 27 26 25 24 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 0 1 1 1 1 1
```

```
if !HaveRASExt() then EndOfInstruction();

Operation

if TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);
    SynchronizeErrors();
    AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} & EL2Enabled() then AArch64.vESBOperation();
    TakeUnmaskedSErrorInterrupts();
```
Extract register extracts a register from a pair of registers.

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

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

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

**EXTR <Wd>, <Wn>, <Wm>, #<lsb>**

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

**EXTR <Xd>, <Xn>, <Xm>, #<lsb>**

- integer \( d = \text{UInt}(Rd); \)
- integer \( n = \text{UInt}(Rn); \)
- integer \( m = \text{UInt}(Rm); \)
- integer \( \text{datasize} = \text{if} \ sf == '1' \ \text{then} \) 64 \( \text{else} \) 32;
- integer \( \text{lsb}; \)

- if \( N != sf \) then UNDEFINED;
- if \( sf == '0' \ & \ \text{imms<5> == '1'} \) then UNDEFINED;
- \( \text{lsb} = \text{UInt}(\text{imms}); \)

### 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 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><strong>ROR (immediate)</strong></td>
<td>( Rn == Rm )</td>
</tr>
</tbody>
</table>

### Operation

- \( \text{bits(datasize)} \ \text{result}; \)
- \( \text{bits(datasize)} \ \text{operand1} = X[n]; \)
- \( \text{bits(datasize)} \ \text{operand2} = X[m]; \)
- \( \text{bits(2*datasize)} \ \text{concat} = \text{operand1:operand2}; \)

\( \text{result} = \text{concat<lsb+datasize-1:lsb>}; \)

\( X[d] = \text{result}; \)
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.
**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  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | Xm  | 0  | 0  | 0  | 1  | 0  | 1  | Xn  | Xd  |

**Integer**

GMI `<Xd>`, `<Xn|SP>`, `<Xm>`

```plaintext
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**

```plaintext
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
HINT

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 0 1 0 1 0 0 0 0 1 1 0 0 1 0 CRm op2 1 1 1 1

System

HINT #<imm>

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_BT;
      // 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":

- 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.
case op of
  when SystemHintOp_YIELD
    Hint_Yield();
  when SystemHintOp_DGH
    Hint_DGH();
  when SystemHintOp_WFE
    if IsEventRegisterSet() then
      ClearEventRegister();
    else
      if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, TRUE);
      if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, TRUE);
      if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, TRUE);
        WaitForEvent();
    endif
  when SystemHintOp_WFI
    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();
    endif
  when SystemHintOp_SEV
    SendEvent();
  when SystemHintOp_SEVL
    SendEventLocal();
  when SystemHintOp_ESB
    if 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_CSB
    ConsumptionOfSpeculativeDataBarrier();
  when SystemHintOp_BTI
    SetBTypeNext('00');
  otherwise // do nothing
HLT

Halt instruction. A HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.

Assembler Symbols

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

Operation

Halt(DebugHalt_HaltInstruction);
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

System

HVC #<imm>

// Empty.

Assembler Symbols

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

Operation

```cpp
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && (!IsSecureEL2Enabled() && IsSecure())) then
  UNDEFINED;

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);
if hvc_enable == '0' then
  UNDEFINED;
else
  AArch64.CallHypervisor(imm16);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 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 | 1 | 1 | 0 | Xm |
| 0 | 0 | 0 | 1 | 0 | 0 | Xn |
|     |     |

Integer

IrG <Xd|SP>, <Xn|SP>{, <Xm>}

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() then
    if GCR_EL1.RRND == '1' then
        rtag = _ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start = RGSR_EL1.TAG;
        bits(4) offset = AArch64.RandomTag();
        rtag = AArch64.ChooseNonExcludedTag(start, offset, exclude);
    else
        rtag = '0000';
bits(64) result = AArch64.AddressWithAllocationTag(operand, 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 0 1 1 0 0 1 1 CRm 1 1 0 1 1 1 1 opc
```

System

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 no memory ordering requirements.

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)*

<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>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>Rn</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>
</tr>
</tbody>
</table>

*opc*
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_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>STADD, STADDL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDADD, LDADDA, LDADDAL, LDADDL
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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.
LDADDDB, LDADDAB, LDADDLB, 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 LDADDLB load with acquire semantics.
- LDADDB and LDADDLB store with release semantics.
- LDADDDB has no memory ordering requirements.

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | A  | R  | 1  | Rs | 0  | 0  | 0  | 0  | 0  | Rn | 0  | Rt |

size opc

LDADDAB (A == 1 && R == 0)

LDADDAB <Ws>, <Wt>, [<Xn|SP>]

LDADDLB (A == 1 && R == 1)

LDADDLB <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;

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>STADDB, STADDLB</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDADDB, LDADDAB, LDADDLB, LDADDLB
**Operation**

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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 **STADDDH, STADDLH**.

**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 0 | A | R | 1 | Rs | 0 0 0 0 0 | 0 | Rn | Rt |
| 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>STADDDH, STADDLH</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDADDAH, LDADDAH, LDADDALH, LDADDLH
**Operation**

\[
\text{bits(64) address; \quad bits(16) value; \quad bits(16) data;}
\]

if \text{HaveMTEExt()} then
  \text{SetNotTagCheckedInstruction(!tag_checked);} \\
\text{value = X[s];}
if n == 31 then
  \text{CheckSPAlignment();}
  \text{address = SP[];}
else
  \text{address = X[n];}
\text{data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);} \\
if t != 31 then
  \text{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.
**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 (Arm v8.3)

<table>
<thead>
<tr>
<th>32-bit (size == 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAPR &lt;Wt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit (size == 11)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAPR &lt;Xt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

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 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**

```
bits(64) address;  
bits(elsize) data;  
constant integer dbytes = elsize DIV 8;  
if HaveMTEExt() then  
    SetNotTagCheckedInstruction(!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.
LDAPRB

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | (1)| (1)| (1)| (1)| 1  | 1  | 0  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |

```plaintext
size
Rs
```

### Integer

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

```plaintext
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;
bids(8) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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**

```
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn  |   |
|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

**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  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Rn  |   |
|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Rs**

```
Integer

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

```
binary(64) address;
binary(16) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | Rn |  |
| size | opc |

32-bit (size == 10)

LDAPUR <Wt>, [<Xn|SP>{, #<simm>}

64-bit (size == 11)

LDAPUR <Xt>, [<Xn|SP>{, #<simm>}

integer scale = UInt(size);
bites(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

```c
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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_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  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

imm9   0  0  Rn  Rt

size   opc

Unscaled offset

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

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

boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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)

<table>
<thead>
<tr>
<th>0 1 0 1 0 1 0</th>
<th>imm9</th>
<th>0 0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

**Unscaled offset**

LDAPURH `<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**

```plaintext
type n = UInt(Rn);
type t = UInt(Rt);

boolean tag_checked = n != 31;
```

**Operation**

```plaintext
if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);

bits(64) address;
b 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.
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  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | x  | 0  | imm9 | 0  | 0  | Rn  | Rt  |

**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

```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

```c
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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

if n == 31 then
    if memop != MemOp_PREFETCH then
        CheckSPAlignment();
    else
        address = X[n];
else
    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  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | x  | 0  | imm9 | 0  | 0  | Rn  | 0  | Rt  |

```

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**

```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
    SetNotTagCheckedInstruction(!tag_checked);

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

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPLalignment();
    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)

<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>0</td>
<td>0</td>
<td>imm9</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>
<td></td>
<td></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>

Unscaled offset

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
   SetNotTagCheckedInstruction(!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.
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/ZXR, 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   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| x   | 0   | 0   | 0   | 0   | 1   | 1   | 0   | (1) | (1) | (1) | (1) | 1   | (1) | (1) | (1) | 1   | (1) | (1) | (1) | (1) | Rn  |    |    |    |    |    |    |    |    |    |
|     | L   | Rs  | o0  | Rt2 |

**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
    SetNotTagCheckedInstruction(!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/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

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
    SetNotTagCheckedInstruction(!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.
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/ZXR, it is impossible for software to observe the presence of the acquire
semantic other than its effect on the arrival at endpoints.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire
semantic other than its effect on the arrival at endpoints.

![Binary representation of LDARH instruction](image)

### No offset

**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

```pseudo
bits(64) address;
bits(16) data;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);
if n == 31 then
    CheckSAPortAlignment();
    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.

---

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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;

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

bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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();

if n == 31 then
    CheckSPAlignment();
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() 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
    SetNotTagCheckedInstruction(!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_ORDEREDATOMIC];
    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.

<table>
<thead>
<tr>
<th>Size</th>
<th>L</th>
<th>Rs</th>
<th>o0</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

No offset

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
    SetNotTagCheckedInstruction(!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.

No offset

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
    SetNotTagCheckedInstruction(!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.
**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 no memory ordering requirements.

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  | 0  | Rn | 0  | 0  | Rn | 0  | 0  | Rn | 0  | 0  | Rn | 0  | 0  |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDCLR, LDCLRA, LDCLRAL, LDCLRL
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 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, STCLRL</td>
<td>A == '0' &amp;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDCLR, LDCLRA, LDCLRAL, LDCLRL
Operation

bits(64) address;
bets(datasize) value;
bets(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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.
LDCLRAB, LDCLRALB, LDCLRB, 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.
- LDCLRB and LDCLRALB store to memory with release semantics.
- LDCLRB has no memory ordering requirements.

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 STCLRB, STCLRLB.

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    | A  | R  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

size opc

LDCLRAB (A == 1 & R == 0)
LDCLRAB <Ws>, <Wt>, [<Xn|SP]>

LDCLRALB (A == 1 & R == 1)
LDCLRALB <Ws>, <Wt>, [<Xn|SP]>

LDCLRB (A == 0 & R == 0)
LDCLRB <Ws>, <Wt>, [<Xn|SP]>

LDCLRLB (A == 0 & R == 1)
LDCLRLB <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>STCLRB, STCLRLB</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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.
LDCLRH, LDCLRAH, LDCLRALH, LDCLRHL

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.
- LDCLRHL and LDCLRALH store to memory with release semantics.
- LDCLRH has no memory ordering requirements.

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)

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>0 0 0 1 0 0</th>
<th>Ws</th>
<th>Wt</th>
<th>Xn</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1 1 0 0</td>
<td>A</td>
<td>R</td>
<td>1</td>
<td>Rs</td>
</tr>
<tr>
<td>size</td>
<td>opc</td>
<td>0 0 0 1 0 0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 HaveMTExe() then
  SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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*

<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>x</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>1</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>R</td>
<td></td>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
</tr>
</tbody>
</table>

LDEOR, LDEORA, LDEORAL, LDEORL
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 HaveMTEEExt() then
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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

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  | A  | R  | 1  | Rs | 0  | 0  | 1  | 0  | 0  | Rn | 0  | 0  | 1  | 0  | 0  | R  | t  |
| 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
    SetNotTagCheckedInstruction(!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.
LDEORH, LDEORAH, LDEORALH, LDEORLH

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 no memory ordering requirements.

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 1 0 0 0 | A | R | 1 | Rs | 0 0 1 0 0 0 | Rn | Rt |

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

#### 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]

if !HaveAtomicExt() then UNDEFINED;

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

AccType ldacctype = if A == '1' &eamp; Rt eamp; '1111' 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;eamp; Rt == '11111'</td>
</tr>
</tbody>
</table>

LDEORH, LDEORAH, LDEORALH, LDEORLH
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!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.
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 0 1 1 0 0 1 1 0 1 1 | imm9 | 0 0 | Xn | Xt |

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], 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_AA64FPSCR_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  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Integer

LDGM <Xt>, [<Xn|SP>]

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
  CheckSParignment();
  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/ZXR, 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 |    |    |
| size | L  | Rs | o0 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

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
  SetNotTagCheckedInstruction(!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/ZXR, 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)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|0 0|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|

**No offset**

`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**

```plaintext
bits(64) address;
bits(8) data;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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/ZXR, 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   | 1   | 1   | 1   | 0   | (1) | (1) | (1) | (1) | (1) | 0   | (1) | (1) | (1) | (1) | (1) | (1) | L   | Rs  | o0  | Rt2 |

No offset

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
  SetNotTagCheckedInstruction(!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
<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>L</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 == 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;
b bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
```
Operation

bits(64) address;
bids(datasize) data1;
bids(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);

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();

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.
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**

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**

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 = FALSE;

**Signed offset**

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

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;
bds(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);

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_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();

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;
  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.
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>Opcode</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101</td>
<td>imm7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = TRUE;

Pre-index

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101</td>
<td>imm7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = FALSE;

Signed offset

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101</td>
<td>imm7</td>
<td></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);
ininteger t = UInt(Rt);
ininteger t2 = UInt(Rt2);
bint(64) offset = LSL(SignExtend(imm7, 64), 2);
boolen tag_checked = wback || n != 31;

**Operation**

bits(64) address;
bits(32) data1;
bits(32) data2;
boolen rt_unknown = FALSE;

if HaveMTExe() then
  SetNotTagCheckedInstruction(!tag_checked);

boolen wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
  Constraint c = ConstraintUnpredictable(Unpredictable_WBOVERLAPL);
  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 wbUnknown = TRUE;  // writeback is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

if t == t2 then
  Constraint c = ConstraintUnpredictable(Unpredictable_LDPOVERLAP);
  assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
  case c of
    when Constraint_UNKNOWN rtUnknown = TRUE;  // result is UNKNOWN
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

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

if !postindex then
  address = address + offset;

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);

if wback then
  if wbUnknown 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 (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

<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</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDR <Wt>, [<Xn|SP>], #<simm>

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>], #<simm>

```java
boolean wback = TRUE;
obolean postindex = TRUE;
integer scale = UInt(size);
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>
<th>1 1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (size == 10)

LDR <Wt>, [<Xn|SP>], #<simm>!

64-bit (size == 11)

LDR <Xt>, [<Xn|SP>], #<simm>!

```java
boolean wback = TRUE;
obolean postindex = FALSE;
integer scale = UInt(size);
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>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>opc</td>
<td></td>
</tr>
</tbody>
</table>


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> 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;
Operation

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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

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();

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
  SetNotTagCheckedInstruction(TRUE);

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
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

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
    SetNotTagCheckedInstruction(FALSE);

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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

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
    SetNotTagCheckedInstruction(!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**

```
0 0 1 1 1 0 0 0 0 1 0 |imm9| 0 1 |Rn|Rt

```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see Architectural Constraints on UNPREDICTABLE behaviors, and particularly LDRH (immediate).

**Pre-index**

```
0 0 1 1 1 0 0 0 0 1 0 |imm9| 1 1 |Rn|Rt

```

**Unsigned offset**

```
0 0 1 1 1 0 0 0 1 0 |imm12| |Rn|Rt

```

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.
Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

**Shared Decode**

```plaintext
gerster n = UInt(Rn);
gster t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

**Operation**

```plaintext
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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

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();

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)

LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]}

Shifted register (option == 011)

LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}

if option<1> == '0' then UNDEFINED; // sub-word index

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

integer n  = UInt(Rn);
integer t  = UInt(Rt);
integer m  = UInt(Rm);
Operation

bits(64) offset = ExtendReg(m, extend_type, 0);
if HaveMTEExt() then
    SetNotTagCheckedInstruction(FALSE);

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.
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</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>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>imm9</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>

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>1</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>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>1</td>
<td>imm9</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>

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>1</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>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>imm12</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>

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.
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**

```plaintext
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

**Operation**

```plaintext
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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

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();

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;
```

**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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

32-bit

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>&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 = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
  SetNotTagCheckedInstruction(FALSE);

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.
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | x  | 0  | imm9|    | 0 | 1 | Rn |    | Rt |

**32-bit (opc == 11)**

LDRSB <Wt>, [<Xn|SP>], #<simm>

**64-bit (opc == 10)**

LDRSB <Xt>, [<Xn|SP>], #<simm>

```java
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  | 0  | 0  | 0  | 1  | x  | 0  | imm9|    | 1 | 1 | Rn |    | Rt |

**32-bit (opc == 11)**

LDRSB <Wt>, [<Xn|SP>], #<simm>!

**64-bit (opc == 10)**

LDRSB <Xt>, [<Xn|SP>], #<simm>!

```java
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  | 0  | 0  | 1  | 1  | x  | 0  | imm12|    | Rn |    | Rt |

```java
```
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.

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 && (wback || n != 31);
if \textbf{HaveMTEExt}() then
\hspace*{1em} SetNotTagCheckedInstruction(!tag_checked);

\textbf{bits}(64) address;
\textbf{bits}(8) data;

boolean \textbf{wb}unknown = FALSE;
boolean \textbf{rt}unknown = FALSE;

if \textbf{memop} == \textbf{MemOp\_LOAD} \&\& wback \&\& n == t \&\& n != 31 then
\hspace*{1em} \textbf{c} = \textbf{ConstrainUnpredictable}(\textbf{Unpredictable\_WBOVERLAPLD});
\hspace*{1em} assert \textbf{c} \textbf{IN} \{\textbf{Constraint\_WBSUPPRESS}, \textbf{Constraint\_UNKNOWN}, \textbf{Constraint\_UNDEF}, \textbf{Constraint\_NOP}\};
\hspace*{1em} case \textbf{c} of
\hspace*{2em} when \textbf{Constraint\_WBSUPPRESS} wback = FALSE; // writeback is suppressed
\hspace*{2em} when \textbf{Constraint\_UNKNOWN} \textbf{wb}unknown = TRUE; // writeback is UNKNOWN
\hspace*{2em} when \textbf{Constraint\_UNDEF} \textbf{UNDEFINED};
\hspace*{2em} when \textbf{Constraint\_NOP} \textbf{EndOfInstruction}();

if \textbf{memop} == \textbf{MemOp\_STORE} \&\& wback \&\& n == t \&\& n != 31 then
\hspace*{1em} \textbf{c} = \textbf{ConstrainUnpredictable}(\textbf{Unpredictable\_WBOVERLAPST});
\hspace*{1em} assert \textbf{c} \textbf{IN} \{\textbf{Constraint\_NONE}, \textbf{Constraint\_UNKNOWN}, \textbf{Constraint\_UNDEF}, \textbf{Constraint\_NOP}\};
\hspace*{1em} case \textbf{c} of
\hspace*{2em} when \textbf{Constraint\_NONE} \textbf{rt}unknown = FALSE; // value stored is original value
\hspace*{2em} when \textbf{Constraint\_UNKNOWN} \textbf{rt}unknown = TRUE; // value stored is UNKNOWN
\hspace*{2em} when \textbf{Constraint\_UNDEF} \textbf{UNDEFINED};
\hspace*{2em} when \textbf{Constraint\_NOP} \textbf{EndOfInstruction}();

if \textbf{n} == 31 then
\hspace*{1em} if \textbf{memop} != \textbf{MemOp\_PREFETCH} then \textbf{CheckSPAlignment}();
\hspace*{1em} \textbf{address} = \textbf{SP}[];
else
\hspace*{1em} \textbf{address} = \textbf{X}[\textbf{n}];
\hspace*{1em} if \textbf{!postindex} then
\hspace*{1em} \textbf{address} = \textbf{address} + \textbf{offset};

\textbf{case memop of}
\hspace*{1em} when \textbf{MemOp\_STORE}
\hspace*{2em} if \textbf{rt}unknown then
\hspace*{3em} \textbf{data} = \textbf{bits}(8) \textbf{UNKNOWN};
\hspace*{2em} else
\hspace*{3em} \textbf{data} = \textbf{X}[\textbf{t}];
\hspace*{2em} \textbf{Mem}[\textbf{address}, 1, AccType\_NORMAL] = \textbf{data};
\hspace*{1em} when \textbf{MemOp\_LOAD}
\hspace*{2em} \textbf{data} = \textbf{Mem}[\textbf{address}, 1, AccType\_NORMAL];
\hspace*{2em} if \textbf{signed} then
\hspace*{3em} \textbf{X}[\textbf{t}] = \textbf{SignExtend}(\textbf{data}, \textbf{regsize});
\hspace*{2em} else
\hspace*{3em} \textbf{X}[\textbf{t}] = \textbf{ZeroExtend}(\textbf{data}, \textbf{regsize});
\hspace*{1em} when \textbf{MemOp\_PREFETCH}
\hspace*{2em} \textbf{Prefetch} (\textbf{address}, t<4:0>);

\hspace*{1em} if \textbf{wback} then
\hspace*{2em} if \textbf{wb}unknown then
\hspace*{3em} \textbf{address} = \textbf{bits}(64) \textbf{UNKNOWN};
\hspace*{2em} elsif \textbf{postindex} then
\hspace*{3em} \textbf{address} = \textbf{address} + \textbf{offset};
\hspace*{2em} if \textbf{n} == 31 then
\hspace*{3em} \textbf{SP}[] = \textbf{address};
\hspace*{2em} else
\hspace*{3em} \textbf{X}[\textbf{n}] = \textbf{address};

\textbf{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.

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

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 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
  SetNotTagCheckedInstruction(!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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | imm9 | 0  | 1  | Rn | Rt |

size        opc

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | imm9 | 1  | 1  | Rn | Rt |

size        opc

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | x  | imm12 |  | Rn |  | Rt |

size        opc
### 32-bit (opc == 11)

LDRSH <Wt>, [<Xn|SP>{, #<pimm}>]

### 64-bit (opc == 10)

LDRSH <Xt>, [<Xn|SP>{, #<pimm}>]

```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](#arch-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

```plaintext
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);
```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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

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 {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();

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.

```
<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>1</td>
<td>x</td>
<td>1</td>
<td>Rm</td>
<td>option</td>
<td>S</td>
<td>1</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>
<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>
```

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>

<table>
<thead>
<tr>
<th>&lt;amount&gt;</th>
<th>Is the index shift amount, optional only when &lt;extend&gt; is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in &quot;S&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>&lt;amount&gt;</td>
</tr>
<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
    SetNotTagCheckedInstruction(!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.
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

```assembly
LDRSW <Xt>, [<Xn|SP>], #<simm>
```

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);

### Pre-index

```assembly
LDRSW <Xt>, [<Xn|SP>, #<simm>]
```

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);

### Unsigned offset

```assembly
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.
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 \(<\text{pimm}>/4\).

**Shared Decode**

```plaintext
text
integer n = \(\text{UInt}(\text{Rn})\);
text
integer t = \(\text{UInt}(\text{Rt})\);
text
boolean tag_checked = wback || n != 31;
```

**Operation**

```plaintext
if \(\text{HaveMTEExt}()\) then
    SetNotTagCheckedInstruction(!tag_checked);

bits(64) address;
bits(32) data;

boolean wb_unknown = FALSE;

if wback && n == t && n != 31 then
    c = \(\text{ConstrainUnpredictable}(\text{Unpredictable_WBOVERLAPLD})\);
    assert c IN \{\text{Constraint_WBSUPPRESS}, \text{Constraint_UNKNOWN}, \text{Constraint_UNDEF}, \text{Constraint_NOP}\};
    case c of
        when \text{Constraint_WBSUPPRESS} wback = FALSE;  // writeback is suppressed
        when \text{Constraint_UNKNOWN} wb unknown = TRUE;  // writeback is UNKNOWN
        when \text{Constraint_UNDEF} UNDEFINED;
        when \text{Constraint_NOP} \text{EndOfInstruction}();

if n == 31 then
    \text{CheckSPAlignment}();
    address = \text{SP}[];
else
    address = \text{X}[n];

if !postindex then
    address = address + offset;

data = \text{Mem}[address, 4, \text{AccType_NORMAL}];
\text{X}[t] = \text{SignExtend}(data, 64);
if wback then
    if wb unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        \text{SP}[] = address;
    else
        \text{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.

---

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | imm19 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

Literal

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
    SetNotTagCheckedInstruction(TRUE);
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.

<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>size opc Rm option S 1 0</td>
</tr>
</tbody>
</table>

64-bit

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>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>#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
    SetNotTagCheckedInstruction(FALSE);

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 no memory ordering requirements.

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  | A  | R  | 1  | Rs | 0  | 0  | 1  | 1  | 0  | 0  | Rn | 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 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>STSET, STSETL</td>
<td>A == '0' &amp; R == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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)

```
  31 30 29 28 27 26 25 24 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
0 0 1 1 1 0 0 0 A R 1 | Rs 0 0 1 1 0 0 | Rn | Rt
```

**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
    SetNotTagCheckedInstruction(!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, LDSETAH, LDSETALH, LDSETLH

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, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- LDSETH has no memory ordering requirements.

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)

<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></td>
<td></td>
<td></td>
<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

LDSETH (A == 1 && R == 0)

LDSETH <Ws>, <Wt>, [<Xn|SP>]

LDSETALH (A == 1 && R == 1)

LDSETALH <Ws>, <Wt>, [<Xn|SP>]

LDSETH (A == 0 && R == 0)

LDSETH <Ws>, <Wt>, [<Xn|SP>]

LDSETLH (A == 0 && R == 1)

LDSETLH <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
    SetNotTagCheckedInstruction(!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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 no memory ordering requirements.

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 | A | R | 1 | Rs | 0 | 1 | 0 | 0 | 0 | Rn | Rt |

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; Rt == '11111'</td>
</tr>
</tbody>
</table>
**Operation**

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_check);

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 no memory ordering requirements.

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)**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| size | opc |
```

**LDSMAXB (A == 1 & R == 0)**

LDSMAXB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXAB (A == 1 & R == 1)**

LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXLB (A == 0 & R == 0)**

LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXLB (A == 0 & R == 1)**

LDSMAXLB <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_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.
- **<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; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 0  | 0  | Rn | 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>

**LDSMAXH (A == 1 && R == 0)**

LDSMAXH <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXAH (A == 1 && R == 1)**

LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]

**LDSMAXALH (A == 0 && R == 0)**

LDSMAXALH <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
    SetNotTagCheckedInstruction(!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.
LDMSIN, LDMSINA, LDMSINAL, LDMSINL

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, LDMSINA and LDMSINAL load from memory with acquire semantics.
- LDMSINL and LDMSINAL store to memory with release semantics.
- LDMSIN has no memory ordering requirements.

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  | A  | R  | 1  |     | Rs  | 0  | 1  | 0  | 1  | 0  | 0  | Rn  |     |     |     |     |     |     |     |     |     |     |     |     |     |

size opc
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;
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>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
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | Rn | 0  | 1  | 0  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  |

<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]

**LDSMINALB (A == 0 && R == 1)**

LDSMINALB <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
    SetNotTagCheckedInstruction(!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.

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 no memory ordering requirements.

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 | Rt |

size opc

LDSMINAH (A == 1 && R == 0)

LDSMINAH <Ws>, <Wt>, [<Xn|SP>]

LDSMINALH (A == 1 && R == 1)

LDSMINALH <Ws>, <Wt>, [<Xn|SP>]

LDSMINH (A == 0 && R == 0)

LDSMINH <Ws>, <Wt>, [<Xn|SP>]

LDSMINLH (A == 0 && R == 1)

LDSMINLH <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>STSMINH, STSMINLH</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
  SetNotTagCheckedInstruction(!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.

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

```c
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

bits(64) address;
bites(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.

Unscaled offset

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
  SetNotTagCheckedInstruction(!tag_checked);

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

if n == 31 then
  CheckSPAlignment();
  address = SP[1];
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.

Unscaled offset

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> == '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;

boolean tag_checked = n != 31;

Operation

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);

bits(64) address;
bits(16) data;
if n == 31 then
  CheckSPAlignment();
  address = SP[1];
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.

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 0 0 0 1 x 0</td>
<td>1 0</td>
<td>Rn</td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit (opc == 11)

LDTRSB <Wt>, [<Xn|SP>{, #<simm}>]

### 64-bit (opc == 10)

LDTRSB <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);

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

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!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] = 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 = 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

```plaintext
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

```plaintext
if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);

bits(64) address;
bits(16) data;
if n == 31 then
  if memop != MemOp_PREFETCH then CheckSParAlignment();
  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.
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.

Unscaled offset

LDTRSW <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);

unpriv_at_el1 = PSTATE.EL == EL1 & EL2Enabled() & 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
    SetNotTagCheckedInstruction(!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];
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.
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 no memory ordering requirements.

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  | 0  | A  | R | 1 | Rs | 0  | 1  | 1  | 0  | 0  | 0  | Rn |    |    |    |    |    |    |
|    | size |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    | 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 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>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
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 0  | Rn | 0  | 0  | 0  | 0  | 0  | A  | R  | 1  | Rs | 0  | 0  |

size opc

**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;
bhits(8) value;
bhits(8) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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)

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>Ws</th>
<th>Wt</th>
<th>Xn</th>
<th>SP</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>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
</tr>
</tbody>
</table>

LDUMAXH (A == 1 & R == 0)

LDUMAXH <Ws>, <Wt>, [<Xn|SP>]

LDUMAXAH (A == 1 & R == 1)

LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]

LDUMAXLH (A == 0 & R == 0)

LDUMAXLH <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;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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  | Rs | 0  | 1  | 1  | 1  | 0  | 0  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |
| size | opc |
32-bit LDUMIN (size == 10 && A == 0 && R == 0)
LDUMIN <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINA (size == 10 && A == 1 && R == 0)
LDUMINA <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINAL (size == 10 && A == 1 && R == 1)
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]

32-bit LDUMINL (size == 10 && A == 0 && R == 1)
LDUMINL <Ws>, <Wt>, [<Xn|SP>]

64-bit LDUMIN (size == 11 && A == 0 && R == 0)
LDUMIN <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINA (size == 11 && A == 1 && R == 0)
LDUMINA <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINAL (size == 11 && A == 1 && R == 1)
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]

64-bit LDUMINL (size == 11 && A == 0 && R == 1)
LDUMINL <Xs>, <Xt>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

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

integer datasiize = 8 <= UInt(size);
integer regsize = if datasiize == 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>STUMIN, STUMINL</td>
<td>A == '0' &amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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)

<table>
<thead>
<tr>
<th>A</th>
<th>R</th>
<th>Rs</th>
<th>Ws</th>
<th>Wt</th>
<th>Xn</th>
<th>SP</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>
</tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LDUMINB (A == 1 && R == 0)

LDUMINB <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;&amp; Rt == '11111'</td>
</tr>
</tbody>
</table>
Operation

bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 no memory ordering requirements.

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  | 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;

```plaintext
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
    SetNotTagCheckedInstruction(!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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 |

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
  SetNotTagCheckedInstruction(!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  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | imm9| 0  | 0  | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size| opc|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Unscaled offset

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

integer n = `UInt`(Rn);
integer t = `UInt`(Rt);
boolean tag_checked = n != 31;

Operation

if `HaveMTEExt`() then
    `SetNotTagCheckedInstruction`(!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

Unscaled offset

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
  SetNotTagCheckedInstruction(!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

```plaintext
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

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

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
    SetNotTagCheckedInstruction(!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_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.

Unscaled offset

LDURSW <Xt>, [<Xn|SP>{, #<simm>}

bits(64) offset = \text{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 = \text{UInt}(Rn);
integer t = \text{UInt}(Rt);

boolean tag_checked = n != 31;

Operation

if \text{HaveMTEExt}() then
    SetNotTagCheckedInstruction(!tag_checked);

bits(64) address;
bits(32) data;

if n == 31 then
    \text{CheckSPAlignment}();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = \text{Mem}[address, 4, \text{AccType\_NORMAL}];
X[t] = \text{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*.

![Address Format](image)

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;
```

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;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

if t == t2 then
    Constraint c = Construnpredictable(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();

if n == 31 then
    CheckSPAlignment();
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() 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
    // 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}]

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;

Assemble 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
  SetNotTagCheckedInstruction(!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</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1 0 0 0 0 1 0 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1</td>
<td>Rn</td>
</tr>
<tr>
<td>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</td>
<td>Rt</td>
</tr>
</tbody>
</table>

No offset

LDXRB <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
    SetNotTagCheckedInstruction(!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_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.
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)(1)(1)(1)(1) 0 (1)(1)(1)(1)(1) 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
    SetNotTagCheckedInstruction(!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_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.
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)`.

| sf | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 0 | Rn | Rd |
|----|---|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|
|    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| op2 |

32-bit (sf == 0)

`LSLV <Wd>, <Wn>, <Wm>`

64-bit (sf == 1)

`LSLV <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.
LSRV

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).

| sf | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 0 | 1 | Rn | Rd |
| op2 |

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

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>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rm</th>
<th>0</th>
<th>Ra</th>
<th>Rn</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>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

MADD <Wd>, <Wn>, <Wm>, <Wa>

64-bit (sf == 1)

MADD <Xd>, <Xn>, < Xm>, <Xa>

```plaintext
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

```plaintext
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 theNZCV flags.

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

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.

```
| sf | 1 1 1 0 0 1 0 1 | hw | imm16 | Rd |
```

**32-bit (sf == 0 && hw == 0x)**

MOVK `<Wd>`, `#<imm>{, LSL #<shift>}`

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

MOVK `<Xd>`, `#<imm>{, LSL #<shift>}`

```plaintext
d = UInt(Rd);
datasize = if sf == '1' then 64 else 32;
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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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
  opc
```

### 32-bit (sf == 0 && hw == 0x)

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

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

```
MOVN <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>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;&amp; hw != '00')</td>
</tr>
<tr>
<td>MOV (inverted wide immediate)</td>
<td>32-bit</td>
<td>! (IsZero(imm16) &amp;&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 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;&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.
MRS

Move System Register allows the PE to read an AArch64 System register into a 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  1 |  1 |  0 |  1 |  0 |  1 |  0 |  0 |  1 |  1 |  o0 |  op1 | CRn | CRm |  op2 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

System

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":
  - 0: 2
  - 1: 3
- `<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);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

System

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');

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 '011 100'
    if !HaveMTEExt() then
      UNDEFINED;
    field = PSTATEField_TCO;
  when '011 110'
    field = PSTATEField_DAIFSet;
  when '011 111'
    field = PSTATEField_DAIFClr;
  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 !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '1') || SCTLR_EL1.UMA == '0') then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' 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>ARMv8.2-UAO</td>
</tr>
<tr>
<td>000</td>
<td>100</td>
<td>PAN</td>
<td>ARMv8.1-PAN</td>
</tr>
<tr>
<td>000</td>
<td>101</td>
<td>SPSele</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>ARMv8.0-SSBS</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>DIT</td>
<td>ARMv8.4-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>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>011</td>
<td>101</td>
<td>RESERVED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
<td>DAIIFSet</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>111</td>
<td>DAIIFClr</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**

```plaintext
case field of
    when PSTATEField_SSBS
        PSTATE.SSBS = CRm<0>;
    when PSTATEField_SP
        PSTATE.SP = CRm<0>;
    when PSTATEField_DAIIFSet
        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_DAIIFClr
        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>;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|
| 01 | 10 | 10 | 10 | 10 | 00 | 01 | 01 | CRn | CRm | op2 | Rt |

System

MSR (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>

AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

t = UInt(Rt);

tsys_op0 = 2 + UInt(o0);
tsys_op1 = UInt(op1);
tsys_op2 = UInt(op2);
tsys_crn = UInt(CRn);
tsys_crm = UInt(CRm);

Assembler Symbols

<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></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.

<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, syscrm, sys_op2, X[t]);
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**.

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

```plaintext
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

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

```plaintext
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

```plaintext
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>;
```

MSUB
Operational information

If PSTATE.DIT is 1:

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

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

System

NOP

// Empty.

Operation

// 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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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**.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</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>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>
</tr>
</tbody>
</table>

32-bit (sf == 0)

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

64-bit (sf == 1)

ORN `<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>MVN</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>
Operation

\[
\text{bits(data size)} \text{ operand1} = X[n]; \\
\text{bits(data size)} \text{ operand2} = \text{ShiftReg}(m, \text{shift type}, \text{shift amount}); \\
\text{operand2} = \text{NOT(operand2)}; \\
\text{result} = \text{operand1 OR operand2}; \\
X[d] = \text{result};
\]

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.
**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).

```
<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>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)**

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

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

ORR <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".

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (bitmask immediate)</td>
<td>Rn == '1111' &amp; MoveWidePreferred(sf, N, imms, immr)</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
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>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</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>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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

ORR \(<Wd>, <Wn>, <Wm>{, <shift> #<amount>}\)

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

ORR \(<Xd>, <Xn>, <Xm>{, <shift> #<amount>}\)

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

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>

- \(<amount>\) is the shift amount for the 32-bit variant, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.
- \(<amount>\) is the shift amount for the 64-bit variant, 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>MOV (register)</strong></td>
<td>(shift == '00' &amp;&amp; \text{imm6} == '000000' &amp;&amp; Rn == '1111')</td>
</tr>
</tbody>
</table>
Operation

bits(datasize) operand1 = X[n];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount);
result = operand1 OR 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.
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  | 0  | 1  | 0  | 0  | Z  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  |

PACD (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
  X[d] = AddPACDA(X[d], SP[]);
else
  X[d] = 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  | 0  | 1  | 0  | 0  | Z  | 0  | 1  | 1  |

### PACDB (Z == 0)

PACDB <Xd>, <Xn|SP>

### PACDZB (Z == 1 && Rn == 11111)

PACDZB <Xd>

```plaintext
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

```plaintext
if source_is_sp then
    X[d] = AddPACDB(X[d], SP[]);
else
    X[d] = AddPACDB(X[d], X[n]);
```

---

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  |

Integer

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>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>1</td>
<td>0</td>
<td>0</td>
<td>Z</td>
<td>0</td>
<td>0</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>
</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>1</td>
<td>1</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>
<td></td>
<td></td>
</tr>
<tr>
<td>CRm</td>
<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>
<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 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.
      - = BTypeCompatible_PACIXSP();
  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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | Z  | 0  | 0  | 1  | Rd |

### PACIB (Z == 0)

PACIB $<$Xd$>$, $<$Xn$|$SP$>$

### PACIZB (Z == 1 && Rn == 11111)

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | x  | 1  | 0  | 1  | x  | 1  | 1  | 1  | 1  | 1  |

CRm  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
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.
      - = 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]);

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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
   size opc
```

Unsigned offset

```
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>`. `<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.

- `<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);
```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(TRUE);

bits(64) address;

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

address = address + offset;

Prefetch(address, t<4:0>);
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.

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

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

bits(64) address = PC[] + offset;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(TRUE);

Prefetch(address, t<4:0>);
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](#).

### Integer

```plaintext
PRFM (<prfop>|#<imm5>), [<Xn|SP>], (<Wm>|<Xm>){, <extend> {<amount>}}
```

if option<1> == '0' then UNDEFINED; // sub-word index

```plaintext
ExtType = 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.
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>

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
ingter n = UInt(Rn);
ingter t = UInt(Rt);
ingter m = UInt(Rm);
```

**Operation**

```plaintext
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetNotTagCheckedInstruction(TRUE);

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.

### Assembler Symbols

| PRFUM (<prfop>|<imm5>|, [<Xn|SP>{, #<simm>}]) |
|-----------------|------------------|
| bits(64) offset = SignExtend(imm9, 64); |

Where:

- `<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.
- `<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);
```
Operation

if HaveMTEExt() then
    SetNotTagCheckedInstruction(TRUE);

bits(64) address;

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

address = address + offset;

Prefetch(address, t<4:0>);
PSB CSYNC

Proiling Synchronization Barrier. This instruction is a barrier that ensures that all existing proiling data for the current PE has been formatted, and proiling buffer addresses have been translated such that all writes to the proiling buffer have been initiated. A following DSB instruction completes when the writes to the proiling buffer have completed.

If the Statistical Proiling Extension is not implemented, this instruction executes as a NOP.

System
(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  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  |    |

CRm op2

System

PSB CSYNC

if !HaveStatisticalProiling() then EndOfInstruction();

Operation

ProilingSynchronizationBarrier();
**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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  |

CRm opc
```

**System**

PSSBB

// No additional decoding required

**Operation**

```c
SpeculativeStoreBypassBarrierToPA();
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
RBIT

Reverse Bits reverses the bit order in a 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 | 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 |

**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**

```plaintext
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**Integer**

RET `<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. Defaults to X30 if absent.

**Operation**

bits(64) target = X[n];

`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, 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)*

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

**RETA** *(M == 0)*

```
RETA
```

**RETB** *(M == 1)*

```
RETB
```

---

```
boolean use_key_a = (M == '0');
if !HavePACEExt() 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);
BranchTo(target, BranchType_RET);
```
Reverse Bytes reverses the byte order in a register.

This instruction is used by the pseudo-instruction REV64.

| sf | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | x | Rn | Rd |
|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| opc | 32-bit (sf == 0 && opc == 10) |
| REV <Wd>, <Wn> |

32-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>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>0</th>
<th>0</th>
<th>0</th>
<th>1</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>
</tr>
</tbody>
</table>

32-bit (sf == 0)

REV16 <Wd>, <Wn>

64-bit (sf == 1)

REV16 <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.
Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.

64-bit

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;
end case;

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;
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.
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 | imm6 | 0 | 0 | 0 | 0 | 1 | Rn | 0 | mask |

sf

Integer

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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 definitions:

- \( d = \text{UInt}(Rd) \)
- \( n = \text{UInt}(Rn) \)
- \( m = \text{UInt}(Rm) \)
- \( \text{datasize} = \text{if sf == '1' then 64 else 32} \)
- \( \text{ShiftType shift_type = DecodeShift(op2)} \)

### 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.
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.

```
<table>
<thead>
<tr>
<th>CRm</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1 0</td>
</tr>
</tbody>
</table>
```

**System**

```
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.

32-bit (sf == 0)

SBC <Wd>, <Wn>, <Wm>

64-bit (sf == 1)

SBC <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>NGC</td>
<td>Rn == '11111'</td>
</tr>
</tbody>
</table>

Operation

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>1</th>
<th>1</th>
<th>1</th>
<th>1</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 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>
<td></td>
<td></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 == '11111'</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 response of this instruction to 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 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.

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

\[
\text{SBFM } <Wd>, <Wn>, #<immr>, #<imms>
\]

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

\[
\text{SBFM } <Xd>, <Xn>, #<immr>, #<imms>
\]

```plaintext
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.

\(<\text{immr}>\) 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>ASR (immediate)</td>
<td>32-bit</td>
<td>imms == '011111'</td>
</tr>
</tbody>
</table>
### 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.
**SDIV**

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.

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

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

SDIV <Wd>, <Wn>, <Wm>

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

SDIV <Xd>, <Xn>, < Xm>

```plaintext
type integer d = UInt(Rd);
type integer n = UInt(Rn);
type integer m = UInt(Rm);
type 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**

```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, FALSE)) / Real(Int(operand2, FALSE)));
X[d] = result<datasize-1:0>;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | 0  | sz| 0  | 0  | 1  | 0  | Rn | 0  | 1  | 1  | 0  | 1 |
```

**SF**

**SETF8 (sz == 0)**

```c
SETF8 <Wn>
```

**SETF16 (sz == 1)**

```c
SETF16 <Wn>

if !HaveFlagManipulateExt() then UNDEFINED;
integer msb = if sz == '1' then 15 else 7;
integer n = UInt(Rn);
```

**Assembler Symbols**

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

**Operation**

```
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.

---

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 1 0 0 1 0 0 0 0 1 0 0 1 1 1 1 1
```

CRm  op2

**System**

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.

```
<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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

System

SEVL

// Empty.

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  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  |

As the 64-bit destination register, encoded in the "Rd" field.

As the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.

As the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

As the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

<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.

System
(Armv8.0-A)

| 31 30 29 28 27 26 25 24 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 | imm16 | 0 0 0 1 1 |

System

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);

if SCR_EL3.SMD == '1' then
  // SMC disabled
  UNDEFINED;
else
  AArch64.CallSecureMonitor(imm16);

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
## SMSUBL

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**.

![Register format](image)

### 64-bit

**SMSUBL <Xd>, <Wn>, <Wm>, <Xa>**

- integer \( d = \text{UInt}(Rd); \)
- integer \( n = \text{UInt}(Rn); \)
- integer \( m = \text{UInt}(Rm); \)
- integer \( a = \text{UInt}(Ra); \)

#### 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 == '1111' )</td>
</tr>
</tbody>
</table>

#### Operation

```plaintext
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 \( \text{PSTATE.DIT} \) 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.
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
  | 0 0 | 1 1 0 1 0 1 0 | Rm | 0 | (1) (1) (1) (1) | Rn | Rd
  | U   | Ra |
```

64-bit

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.
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.

---

System

SSBB

// No additional decoding required

Operation

SpeculativeStoreBypassBarrierToVA();
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  | 1  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| imm9 | 0  | 1  | Xn | Xt |
```

Post-index

ST2G <Xt|SP>, [<Xn|SP>], #<simm>

```java
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 |
```

Pre-index

ST2G <Xt|SP>, [<Xn|SP>], #<simm>

```java
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

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 |
```

Signed offset

ST2G <Xt|SP>, [<Xn|SP>{, #<simm>}

```java
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**

```plaintext
bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
SetNotTagCheckedInstruction(TRUE);

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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 0  | 0  | 1  |     | imm9| 0  | 1  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

#### Post-index

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  | 0  | 0  | 1  |     | imm9| 1  | 1  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

#### Pre-index

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  | 0  | 0  | 1  |     | imm9| 1  | 0  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

#### Signed offset

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

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**

```plaintext
bits(64) address;
SetNotTagCheckedInstruction(TRUE);
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];
bjets(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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 4*A<7:4>+3:4*A<7:4>.
This instruction is UNDEFINED at EL0.
This instruction generates an Unchecked access.
If ID_AA64PPR1_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 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Xn | Xt |

Integer

STGM <<Xt>>, [<Xn|SP>]

t = UInt(Xt);
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 * (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 = data<(index*4)+3:index*4>;
  AArch64_MemTag[address, AccType_NORMAL] = tag;
  address = address + TAG_GRANULE;
  index = index + 1;
STGP

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  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 1   | 0   | 1   | 0   | 0   | 0   | 1   | 0   | simm7 | Xt2 | Xn | Xt |

Post-index

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  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 1   | 0   | 1   | 0   | 1   | 0   | simm7 | Xt2 | Xn | Xt |

Pre-index

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  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0   | 1   | 1   | 0   | 1   | 0   | 1   | 0   | 0   | simm7 | Xt2 | Xn | Xt |

Signed offset

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 = 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.

<imm> 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

\[
\begin{align*}
\text{bits}(64) & \quad \text{address;} \\
\text{bits}(64) & \quad \text{data1;} \\
\text{bits}(64) & \quad \text{data2;} \\
\text{SetNotTagCheckedInstruction}(\text{TRUE}); \\
\text{if } n == 31 \text{ then} & \\
& \quad \text{CheckSPAlignment}(); \\
& \quad \text{address} = \text{SP[]} ; \\
\text{else} & \\
& \quad \text{address} = \text{X}[n] ; \\
\text{data1} = \text{X}[t]; \\
\text{data2} = \text{X}[t2]; \\
\text{if } \text{!postindex} \text{ then} & \\
& \quad \text{address} = \text{address} + \text{offset}; \\
\text{Mem}[\text{address}, 8, \text{AccType_NORMAL}] = \text{data1}; \\
\text{Mem}[\text{address}+8, 8, \text{AccType_NORMAL}] = \text{data2}; \\
\text{AArch64.MemTag}[\text{address, AccType_NORMAL}] = \text{AArch64.AllocationTagFromAddress}(\text{address}); \\
\text{if } \text{writeback} \text{ then} & \\
& \quad \text{if } \text{postindex} \text{ then} \\
& \quad \quad \text{address} = \text{address} + \text{offset}; \\
& \quad \quad \text{if } n == 31 \text{ then} \\
& \quad \quad \quad \text{SP[]} = \text{address}; \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{X}[n] = \text{address}; \\
\end{align*}
\]
STLLR

Store LORelase 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 LORelase. 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
<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>
<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>1</td>
<td>x</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>(1)</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>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
  SetNotTagCheckedInstruction(!tag_checked);
if n == 31 then
  CheckSAPermission();
  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)

![Register Encoding](image)

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
  SetNotTagCheckedInstruction(!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 )

<table>
<thead>
<tr>
<th>0 1 0 0 1 0 0 1 0 0</th>
<th>0 1 0 (1) (1) (1) (1) 0 (1) (1) (1) (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>L             Rs          o0       Rt2</td>
</tr>
</tbody>
</table>

No offset  

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
    SetNotTagCheckedInstruction(!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>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>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>1</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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>L</td>
<td>Rs</td>
<td>o0</td>
<td>Rt2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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)

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

64-bit (size == 11)

STLR <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
    SetNotTagCheckedInstruction(!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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1)| (1) |

No offset

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
  SetNotTagCheckedInstruction(!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.

### 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

```plaintext
bits(64) address;
bits(16) data;
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | imm9| 0  | 0  | Rn | Rt |

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
   SetNotTagCheckedInstruction(!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)

<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>imm9</td>
<td>0</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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unscaled offset

STLURB <Wt>, [<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.
<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 \text{HaveMTEExt}() then
    \text{SetNotTagCheckedInstruction}(!tag_checked);

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

if n == 31 then
    \text{CheckSPAlignment}();
    address = SP[];
else
    address = X[n];

address = address + offset;

data = X[t];
\text{Mem}[address, 1, \text{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.
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 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  | 0  | 0  | 0  | 0  | 0  |

Unscaled offset

STLURH <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
    SetNotTagCheckedInstruction(!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*.

<table>
<thead>
<tr>
<th>sz</th>
<th>0 0 1 0 0 0 0 0</th>
<th>Rs</th>
<th>1</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

STLP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>]{,#0}

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

STLP <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 != 31;

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.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<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.

<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(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);

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 rt_unknown = TRUE; // address is UNKNOWN
    when Constraint_NONE rt unknown = FALSE; // address is original base
    when Constraint_UNDEF UNDEFINED;
    when Constraint_NOP EndOfInstruction();

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() 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_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.
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 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  | 0 |

size

L

Rs

1

(1)

(1)

(1)

(1)

Rn

Rt

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;

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

bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);
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();
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*.

<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 0 0 0 0 1 (1) (1) (1) (1)</td>
</tr>
</tbody>
</table>

No offset

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;

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.
bits(64) address;
b CELtbs(8) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    Set1NotTagCheckedInstruction(!tag_checked);

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();

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_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.
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  |
|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| size |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      | Rs  | 1   | (1) | (1) | (1) | (1) |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
|      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |

No offset

STLXRH <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;

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

bits(64) address;
bits(16) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);

if s == t then
  Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
  assert c IN {ConstraintUNKNOWN, ConstraintNONE, ConstraintUNDEF, ConstraintNOP};
  case c of
    when ConstraintUNKNOWN rt_unknown = TRUE;  // store UNKNOWN value
    when ConstraintNONE rt_unknown = FALSE;  // store original value
    when ConstraintUNDEF UNDEFINED;
    when ConstraintNOP   EndOfInstruction();
  end;

if s == n && n != 31 then
  Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
  assert c IN {ConstraintUNKNOWN, ConstraintNONE, ConstraintUNDEF, ConstraintNOP};
  case c of
    when ConstraintUNKNOWN rn_unknown = TRUE;  // address is UNKNOWN
    when ConstraintNONE rn_unknown = FALSE;  // address is original base
    when ConstraintUNDEF UNDEFINED;
    when ConstraintNOP   EndOfInstruction();
  end;

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, AccTypeORDEREDATOMIC] = 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.

```
| 31 30 29 28 27 26 25 24 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              | imm7           | Rt2            | Rn              |
```

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

```plaintext
t = UInt(Rn);
integer scale = 2 + UInt(opc<1>);
n = UInt(Rt);
t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
data = 8 << scale;
offset = LSL(SignExtend(imm, 64), scale);
tag_checked = n != 31;
```
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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  | 1  | 0  | imm7| Rt2 | Rn | Rt |
| opc| L |

**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  | imm7| Rt2 | Rn | Rt |
| opc| L |

**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 = 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| x  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | imm7| Rt2 | Rn | Rt |
| opc| L |

**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**

<\texttt{Wt1}> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<\texttt{Wt2}> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<\texttt{Xt1}> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<\texttt{Xt2}> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

<\texttt{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 \(<\text{imm}>/4\).

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 \(<\text{imm}>/4\).

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 \(<\text{imm}>/8\).

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 \(<\text{imm}>/8\).

**Shared Decode**

```plaintext
text
integer n = \texttt{UInt}(Rn);
text
integer t = \texttt{UInt}(Rt);
text
integer t2 = \texttt{UInt}(Rt2);
text
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
text
integer scale = 2 + \texttt{UInt}(opc<1>);
text
integer datasize = 8 << scale;
text
bits(64) offset = \texttt{LSL}(\texttt{SignExtend}(imm7, 64), scale);
text
boolean tag_checked = wback || n != 31;
text
```
Operation

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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();

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

32-bit (size == 10)

\[
\text{STR}<Wt>, [<Xn|SP>], <\text{simm}>
\]

64-bit (size == 11)

\[
\text{STR}<Xt>, [<Xn|SP>], <\text{simm}>
\]

boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = \text{UInt}(size);
bits(64) offset = \text{SignExtend}(\text{imm9}, 64);

Pre-index

32-bit (size == 10)

\[
\text{STR}<Wt>, [<Xn|SP>], <\text{simm}>
\]

64-bit (size == 11)

\[
\text{STR}<Xt>, [<Xn|SP>], <\text{simm}>
\]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = \text{UInt}(size);
bits(64) offset = \text{SignExtend}(\text{imm9}, 64);

Unsigned offset

32-bit (size == 10)

\[
\text{STR}<Wt>, [<Xn|SP>], <\text{simm}>
\]

64-bit (size == 11)

\[
\text{STR}<Xt>, [<Xn|SP>], <\text{simm}>
\]

boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = \text{UInt}(size);
bits(64) offset = \text{SignExtend}(\text{imm9}, 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

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

ticknter datsize = 8 << scale;
boolean tag_checked = wback || n != 31;
Operation

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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

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();

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.

```
1 30 29 28 27 26 25 24 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
```

**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
    SetNotTagCheckedInstruction(FALSE);

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

```
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 0 0

boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

### Pre-index

```
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 0 0

boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

### Unsigned offset

```
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 0 0

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.
Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the “imm12” field.

**Shared Decode**

```plaintext
type integer n = UInt(Rn);
type integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

**Operation**

```plaintext
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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

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();

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.

| 31 30 29 28 27 26 25 24 23 22 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 | Rm | option | S | 1 0 | Rn | Rt |

size  opc

Extended register (option != 011)

STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]

Shifted register (option == 011)

STRB <Wt>, [<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.

<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

integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
Operation

bits(64) offset = ExtendReg(m, extend_type, 0);
if HaveMTEExt() then
    SetNotTagCheckedInstruction(FALSE);

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  | 0  | 0  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   | 1 |   |   |   |   |   |   |   |   |   |   |   |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm9</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

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  | 0  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   | 1 |   |   |   |   |   |   |   |   |   |   |   |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

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  | 0  | 0  | 0  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | 1 |   |   |   |   |   |   |   |   |   |   |   |

<table>
<thead>
<tr>
<th>size</th>
<th>opc</th>
<th>imm12</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</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 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.
- <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 &lt;pimm&gt;/2.

**Shared Decode**

```c
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;
```

**Operation**

```c
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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

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();

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.

<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>Rm</td>
<td>option</td>
<td>S</td>
<td>1</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>
<td></td>
</tr>
</tbody>
</table>

**32-bit**

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><code>option</code></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><code>S</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>#1</td>
</tr>
</tbody>
</table>

### Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```
Operation

bits(64) offset = \texttt{ExtendReg}(m, extend\_type, shift);
if \texttt{HaveMTEExt}() then
  \texttt{SetNotTagCheckedInstruction}(FALSE);

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

if n == 31 then
  \texttt{CheckSPAlignment}();
  address = \texttt{SP}[];
else
  address = \texttt{X}[n];

address = address + offset;
data = \texttt{X}[t];
\texttt{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.
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
    SetNotTagCheckedInstruction(!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](#).

<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>0</td>
<td>0</td>
<td>1</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rt</td>
<td></td>
</tr>
</tbody>
</table>

**Unscaled offset**

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**

```plaintext
global

t, n

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**

```plaintext
if HaveMTEExt() then
  SetNotTagCheckedInstruction(!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.

### Unscaled offset

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

```plaintext
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

```plaintext
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

bits(64) address;
bits(16) data;
if n == 31 then
    CheckSPL Alignment();
    address = SP[];
else
    address = X[n];
address = address + offset;
data = X[t];
Mem[address, 2, 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.
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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | x  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | imm9| 0  | 0  | Rn | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

size opc

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
  SetNotTagCheckedInstruction(!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  | imm9|     | 0  | 0  | Rn  |     | Rt  |

size opc

Unscaled offset

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
    SetNotTagCheckedInstruction(!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.

<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>STURH &lt;Wt&gt;, [&lt;Xn</td>
</tr>
</tbody>
</table>

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
    SetNotTagCheckedInstruction(!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 \textit{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 \textit{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}]

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;

For information about the \textbf{CONSTRAINED UNPREDICTABLE} behavior of this instruction, see \textit{Architectural Constraints on UNPREDICTABLE behaviors}, and particularly \textit{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.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<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.

<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(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!tag_checked);
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();
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() 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;

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;
bids(elsize) data;
constant integer dbytes = elsize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);
fi

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();
    end
fi

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();
    end
fi

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

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t];
endif

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);
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.
**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*.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Rs | 0  | (1) | (1) | (1) | (1) | Rn | Rt |

size L o0 Rt2

**No offset**

`STXRB <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;

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:
  - 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;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

if s == t then
    Constraint c = ConstrUncp(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 = ConstrUncp(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();

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_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.
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.

| 31 30 29 28 27 26 25 24 23 22 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 | Rs              | Rn              | Rt              |
| size            | 0 [1] [1] [1]   | [1]            | [1]            |
| 0               | o0              | Rt2             |

No offset

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;

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;
bits(16) data;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

if s == t then
    Constraint c = ConstraintUnpredictable(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 = ConstraintUnpredictable(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();
    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)

```assembly
0 1 1 1 0 0 1 1 1 imm9, 0 1 Xn Xt
```

Post-index
STZ2G <Xt|SP>, [<Xn|SP>], #=>imm

```
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)

```assembly
0 1 1 1 0 0 1 1 1 imm9, 1 1 Xn Xt
```

Pre-index
STZ2G <Xt|SP>, [<Xn|SP>], #=>imm

```
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)

```assembly
0 1 1 1 0 0 1 1 1 imm9, 1 0 Xn Xt
```

Signed offset
STZ2G <Xt|SP>, [<Xn|SP>{, #=>imm}]

```
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**

```c
bits(64) address = if t == 31 then SP[] else X[t];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
SetNotTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);
Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);
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)

<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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>mm9</td>
<td>0</td>
<td>1</td>
<td>Xn</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>
</tr>
</tbody>
</table>

Post-index

STZG <Xt|SP>, [<Xn|SP>], #<simm>

integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(mm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;

Pre-index
(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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>mm9</td>
<td>1</td>
<td>1</td>
<td>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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-index

STZG <Xt|SP>, [<Xn|SP>], #<simm>!

integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(mm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;

Signed offset
(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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>mm9</td>
<td>1</td>
<td>0</td>
<td>Xn</td>
<td>Xt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

Signed offset

STZG <Xt|SP>, [<Xn|SP>], #<simm>

integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(mm9, 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**

```plaintext
bits(64) address;
SetNotTagCheckedInstruction(TRUE);
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n];
if !postindex then
    address = address + offset;
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 $ID_AA64FPRL1_EL1.MTE$ != 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 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Xn | Xt |

Integer

STZGM $<Xt>$, $<[Xn|SP>]$

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
    CheckSPAlignment();
    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;
**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);
integerdatasize = 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>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>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);
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.

```
 31 30 29 28 27 26 25 24 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 | 1 | 0 | sh | imm12 | Rn | Rd |
```

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

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

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

```%
SUB <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>
```

**Operation**

```%
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)**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>shift</th>
<th>0</th>
<th>Rm</th>
<th>imm6</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>
</tr>
</tbody>
</table>

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

\[
\text{SUB } <Wd>, <Wn>, <Wm>\{, <shift> #<amount>\}
\]

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

\[
\text{SUB } <Xd>, <Xn>, <Xm>\{, <shift> #<amount>\}
\]

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

**Asmbl Symbol**

- `<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>NEG (shifted register)</strong></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.
SUBG

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | uimm6 | [0](0) | uimm4 | Xn | Xd |

op3

Integer

SUBG &lt;Xd|SP&gt;, &lt;Xn|SP&gt;, #&lt;uimm6&gt;, #&lt;uimm4&gt;

    integer d = UInt(Xd);
    integer n = UInt(Xn);
    bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);

Assembler Symbols

&lt;Xd|SP&gt;  Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.

&lt;Xn|SP&gt;  Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.

&lt;uimm6&gt; Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.

&lt;uimm4&gt; 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() then
        rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
    else
        rtag = "0000';
    (result, -) = AddWithCarry(operand1, NOT(offset), '1');
    result = AArch64.AddressWithAllocationTag(result, rtag);
    if d == 31 then
        SP[] = result;
    else
        X[d] = result;
SUBP

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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 1  | 1  | 0  | 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  |

| <Xd> | <Xn|SP> | <Xm|SP> |
|------|------|------|

integer \(d = \text{UInt}(Xd);\)  
integer \(n = \text{UInt}(Xn);\)  
integer \(m = \text{UInt}(Xm);\)

**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**

\[
\begin{align*}
\text{bits}(64) \text{ operand1} &= \text{if } n == 31 \text{ then } \text{SP}[] \text{ else } X[n]; \\
\text{bits}(64) \text{ operand2} &= \text{if } m == 31 \text{ then } \text{SP}[] \text{ else } X[m]; \\
\text{operand1} &= \text{SignExtend}(\text{operand1}<55:0>, 64); \\
\text{operand2} &= \text{SignExtend}(\text{operand2}<55:0>, 64); \\
\text{bits}(64) \text{ result}; \\
\text{operand2} &= \text{NOT}(\text{operand2}); \\
(\text{result}, -) &= \text{AddWithCarry}(\text{operand1}, \text{operand2}, '1'); \\
X[d] &= \text{result};
\end{align*}
\]
**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)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |   | Xn | 0 | 0 | 0 | 0 | 0 |   | Xm | 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><strong>CMPP</strong></td>
<td>S == '1' &amp; Xd == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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).

<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>1</th>
<th>0</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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit (sf == 0)

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

64-bit (sf == 1)

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

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>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' (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: 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>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

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

SUBS <Xd>, <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>** 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><strong>CMP (immediate)</strong></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;
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**.

<table>
<thead>
<tr>
<th>sf</th>
<th>1</th>
<th>1</th>
<th>0</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>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)**

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

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

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

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

**Alias Conditions**

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

```
1 1 0 1 0 0 0 0 | imm16 | 0 0 0 0 1
```

**System**

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);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 no memory ordering requirements.

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

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

**Integer**

(Arm*v8.1*)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>x</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>A</th>
<th>R</th>
<th>1</th>
<th>Rs</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<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>
</tr>
</tbody>
</table>
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
    SetNotTagCheckedInstruction(!tag_checked);

if n == 31 then
    CheckSParAlignment();
    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 no memory ordering requirements.

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 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

size

**SWPAB (A == 1 && R == 0)**

SWPAB <Ws>, <Wt>, [<Xn|SP>]

**SWPALB (A == 1 && R == 1)**

SWPALB <Ws>, <Wt>, [<Xn|SP>]

**SWPB (A == 0 && R == 0)**

SWPB <Ws>, <Wt>, [<Xn|SP>]

**SWPLB (A == 0 && R == 1)**

SWPLB <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;
b(8) data;
bits(8) store_value;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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);
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 no memory ordering requirements.

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  | A  | R  | 1  | 0  | 0  | 0  | 0  | Rn | R   |

**size**

**SWPAH (A == 1 && R == 0)**

SWPAH <W>, <W>, [<Xn|SP>]

**SWPALH (A == 1 && R == 1)**

SWPALH <W>, <W>, [<Xn|SP>]

**SWPH (A == 0 && R == 0)**

SWPH <W>, <W>, [<Xn|SP>]

**SWPLH (A == 0 && R == 1)**

SWPLH <W>, <W>, [<Xn|SP>]

if !HaveAtomicExt() then UNDEFINED;

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

`AccType ldacctype = if A == '1' && Rt != '1111' then AccType.ORDEREDATOMICRW else AccType.ATOMICRW;`

`AccType stacctype = if R == '1' then AccType.ORDEREDATOMICRW else AccType.ATOMICRW;`

boolean tag_checked = n != 31;

### Assembler Symbols

- **<W>** 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
    SetNotTagCheckedInstruction(!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);
SYS

System instruction. For more information, see \textit{Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions} for the encodings of System instructions.

This instruction is used by the aliases \texttt{AT}, \texttt{CFP}, \texttt{CPP}, \texttt{DC}, \texttt{DVP}, \texttt{IC}, and \texttt{TLBI}.

\begin{center}
\begin{tabular}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{tabular}
\end{center}

\texttt{L}

egin{enumerate}
\item \texttt{1 1 0 1 0 1 0 0} \texttt{op1} \texttt{CRn} \texttt{CRm} \texttt{op2} \texttt{Rt}
\end{enumerate}

\textbf{System}

\texttt{SYS \#<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}}

\texttt{AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);
}

\begin{verbatim}
integer t = UInt(Rt);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
\end{verbatim}

\textbf{Assembler Symbols}

- \texttt{<op1>} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- \texttt{<Cn>} is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" 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 optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

\textbf{Alias Conditions}

\begin{tabular}{|c|l|}
\hline
\textbf{Alias} & \textbf{Is preferred when} \\
\hline
\texttt{AT} & \texttt{CRn == '0111'} \&\& \texttt{CRm == '100x'} \&\& \texttt{SysOp(op1,'0111',CRm,op2)} == \texttt{Sys_AT} \\
\texttt{CFP} & \texttt{op1 == '011'} \&\& \texttt{CRn == '0111'} \&\& \texttt{CRm == '0011'} \&\& \texttt{op2 == '100'} \\
\texttt{CPP} & \texttt{op1 == '011'} \&\& \texttt{CRn == '0111'} \&\& \texttt{CRm == '0011'} \&\& \texttt{op2 == '101'} \\
\texttt{DC} & \texttt{CRn == '0111'} \&\& \texttt{SysOp(op1,'0111',CRm,op2)} == \texttt{Sys_DC} \\
\texttt{DVP} & \texttt{op1 == '011'} \&\& \texttt{CRn == '0111'} \&\& \texttt{CRm == '0011'} \&\& \texttt{op2 == '101'} \\
\texttt{IC} & \texttt{CRn == '0111'} \&\& \texttt{SysOp(op1,'0111',CRm,op2)} == \texttt{Sys_IC} \\
\texttt{TLBI} & \texttt{CRn == '1000'} \&\& \texttt{SysOp(op1,'1000',CRm,op2)} == \texttt{Sys_TLBI} \\
\hline
\end{tabular}

\textbf{Operation}

\texttt{AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, X[t]);}

\begin{center}
\textit{Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00}
\end{center}

\begin{center}
\textit{Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.}
\end{center}
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.

System

\texttt{SYSL}\ <\texttt{Xt}, \#,\texttt{op1}, \texttt{Cn}, \texttt{Cm}, \#,\texttt{op2}>

\texttt{AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

t = \texttt{UInt}(Rt);

t = \texttt{UInt}(op1);

t = \texttt{UInt}(op2);

t = \texttt{UInt}(CRn);

t = \texttt{UInt}(CRm);

Assembler Symbols

- \texttt{Xt} is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
- \texttt{op1} is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
- \texttt{Cn} is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" 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.

Operation

\texttt{X[t] = AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2);}

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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**

\[
\begin{array}{ccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 1 & Zm & 0 & 0 & 1 & 1 & 0 & 0 & Zn & Zd
\end{array}
\]

SVE

\[
\text{TBL } \langle Zd \rangle.\langle T \rangle, \{ \langle Zn \rangle.\langle T \rangle \}, \langle Zm \rangle.\langle T \rangle
\]

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

\[
\text{integer esize} = 8 \ll \text{UInt}(\text{size});
\]

\[
\text{integer } n = \text{UInt}(Zn);
\]

\[
\text{integer } m = \text{UInt}(Zm);
\]

\[
\text{integer } d = \text{UInt}(Zd);
\]

\[
\text{boolean double_table} = \text{FALSE};
\]

**SVE2**

\[
\begin{array}{ccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & \text{size} & 1 & Zm & 0 & 0 & 1 & 0 & 1 & 0 & Zn & Zd
\end{array}
\]

SVE2

\[
\text{TBL } \langle Zd \rangle.\langle T \rangle, \{ \langle Zn1 \rangle.\langle T \rangle, \langle Zn2 \rangle.\langle T \rangle \}, \langle Zm \rangle.\langle T \rangle
\]

\[
\text{if } \neg \text{HaveSVE2}() \text{ then UNDEFINED;}
\]

\[
\text{integer esize} = 8 \ll \text{UInt}(\text{size});
\]

\[
\text{integer } n = \text{UInt}(Zn);
\]

\[
\text{integer } m = \text{UInt}(Zm);
\]

\[
\text{integer } d = \text{UInt}(Zd);
\]

\[
\text{boolean double_table} = \text{TRUE};
\]

**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>\text{size}</th>
<th>\text{&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.
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.

14-bit signed PC-relative branch offset

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

<table>
<thead>
<tr>
<th>&lt;R&gt;</th>
<th>Is a width specifier, encoded in “b5”:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(b5)</td>
</tr>
<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);

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
TBZ

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>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>b5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>imm14</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14-bit signed PC-relative branch offset

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);

Assemble Symbols

- <R> Is a width specifier, encoded in “b5”:

<table>
<thead>
<tr>
<th>b5</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>

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

(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  | 1  | 0  | 1  | 1  | imm16 | 0  | 0  | 0  | 0  |

System

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
(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  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

System

TCOMMIT

if !HaveTME() then UNDEFINED;

Operation

CheckTMEEnabled();

if TSTATE.depth == 0 then
  UNDEFINED;
if TSTATE.depth == 1 then
  CommitTransactionWrites();
  ClearExclusiveLocal(ProcessorID());
TSTATE.depth = TSTATE.depth - 1;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If ARMv8.4-Trace is not implemented, this instruction executes as a NOP.

System
(Armv8.4)

if !HaveSelfHostedTrace() then EndOfInstruction();

Operation

TraceSynchronizationBarrier();
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
(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  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | Rt |

System

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.EL of
  when EL0
    IsEL1Regime = 51TranslationRegime() == 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
  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();

TSTATE.depth = TSTATE.depth + 1;
X[t] = Zeros(64);
TTEST

This instruction writes the depth of the transaction to the destination register, or the value 0 otherwise.

System
(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>0</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

System

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();

\[ X[t] = (TSTATE.depth)[63:0]; \]
**UBFM**

Unaligned 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 and above the bitfield are set to zero.

This instruction is used by the aliases [LSL (immediate)], [LSR (immediate)], [UBFIZ], [UBFX], [UXTB], and [UXTH].

<table>
<thead>
<tr>
<th>sf</th>
<th>1</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)**

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

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

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

```plaintext
d = \text{UInt}(Rd);
n = \text{UInt}(Rn);

\text{integer} \text{datasize} = \text{if sf} == '1' \text{then 64 else 32};

\text{integer R};
\text{integer} wmask, tmask;

\text{bits(datasize) wmask};
\text{bits(datasize) tmask};

\text{if sf} == '1' \& \& N \neq '1' \text{then UNDEFINED};
\text{if sf} == '0' \& \& (N \neq '0' || \text{immr<5>} \neq '0' || \text{imms<5>} \neq '0') \text{then UNDEFINED};

R = \text{UInt}(immr);
(wmask, tmask) = \text{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>LSL (immediate)</td>
<td>32-bit</td>
<td>\text{imms} != '011111' &amp; &amp; \text{imms + 1} == \text{immr}</td>
</tr>
<tr>
<td>LSL (immediate)</td>
<td>64-bit</td>
<td>\text{imms} != '111111' &amp; &amp; \text{imms + 1} == \text{immr}</td>
</tr>
<tr>
<td>LSR (immediate)</td>
<td>32-bit</td>
<td>\text{imms} == '011111'</td>
</tr>
</tbody>
</table>
Alias | Of variant | Is preferred when
--- | --- | ---
LSR (immediate) | 64-bit | imms == '111111'
UBFIZ | | \( \text{UInt}(\text{imms}) < \text{UInt}(\text{immr}) \)
UBFX | | \( \text{BFXPreferred}(sf, \text{opc}<1>, \text{imms}, \text{immr}) \)
UXTB | | \( \text{immr} == '000000' \&\& \text{imms} == '000111' \)
UXTH | | \( \text{immr} == '000000' \&\& \text{imms} == '001111' \)

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.
Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanentlyUndefined in the Armv8-A architecture.

Integer

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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

![Register Format](image)

32-bit (sf == 0)

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
UDIV <Xd>, <Xn>, < Xm>
```

### 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

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

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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**.

```
<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>0</td>
<td>1</td>
<td>0</td>
<td>1</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>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>Ra</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>o0</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>
<td></td>
<td></td>
<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>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>
</tr>
</tbody>
</table>
```

**64-bit**

**UMADDL** `<Xd>`, `<Wn>`, `<Wm>`, `<Xa>`

```plaintext
nen = UInt(Rn);
n = UInt(Rm);
n = UInt(Ra);
n = UInt(Rd);
```

**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**

```plaintext
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>;
```

**Operational information**

If PSTATE.DIT is 1:

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

---

### 64-bit

**UMSUBL** `<Xd>`, `<Wn>`, `<Wm>`, `<Xa>`

```plaintext
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><strong>UMNEGL</strong></td>
<td>Ra == '11111'</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
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>;
```

**Operational information**

If PSTATE.DIT is 1:

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

| 31 30 29 28 27 26 25 24 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 | 1 | 1 | 0 | Rm | 0 | (1) | (1) | (1) | (1) | Rn | Rd |

64-bit

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

```plaintext
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.
**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.**

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

CRm  op2
```

**System**

WFE

```plaintext
// Empty.
```

**Operation**

```plaintext
if IsEventRegisterSet() then
    ClearEventRegister();
else
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        AArch64.CheckForWFxTrap(EL1, TRUE);
    if PSTATE.EL IN (EL0, EL1) && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch64.CheckForWFxTrap(EL2, TRUE);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
        // Check for traps described by the Secure Monitor.
        AArch64.CheckForWFxTrap(EL3, TRUE);
    WaitForEvent();
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
WFI

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.

```
31 30 29 28 27 26 25 24 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 0 0 0 | 0 1 1 1 1 1 1
CRm  op2
```

System

WFI

// Empty.

Operation

```plaintext
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();
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

System

if !HaveFlagFormatExt() then UNDEFINED;

Operation

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

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

```
31 30 29 28 27 26 25 24 23 22 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   1   1   1   1   1   0
       Rd
```

XPACD (D == 1)

XPACD <Xd>

XPACI (D == 0)

XPACI <Xd>

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

System

XPACLRI

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**

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.

System

YIELD

// Empty.

Operation

Hint_Yield();

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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).
CMTST: 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).
FCMG (register): Floating-point Compare Greater than (vector).
FCMG (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).
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).
FCVTLS, FCVTLS2: 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 (vector): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).
FCVTMU (scalar): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).
FCVTN, FCVTN2: Floating-point Convert to lower precision Narrow (vector).
FCVTNS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to even (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).
FCVTNS (scalar, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).
FCVTNS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTNS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
FCVTNS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTNS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).
FCVTNR (vector): Floating-point Convert to lower precision Narrow, rounding to odd (vector).
FCVTZS (vector): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTZS (vector): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTZS (vector): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).
FCVTZU (vector): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FCVTZU (vector): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FCVTZU (vector): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FCVTZU (vector): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).
FDT (vector): Floating-point Divide (vector).
FDT: Floating-point Divide (scalar).
FDIV: Floating-point Divide (scalar).
FDIV: Floating-point Divide (vector).
FCVTJS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.
FMADD: Floating-point fused Multiply-Add (scalar).
FMAX (vector): Floating-point Maximum (vector).
FMAX: Floating-point Maximum (scalar).
FMAXNM (vector): Floating-point Maximum Number Pairwise (vector).
FMAXNM (scalar): Floating-point Maximum Number (scalar).
FMAXNM: Floating-point Maximum Number (vector).
FMAXNM: 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.

FMIN (scalar): Floating-point Minimum (scalar).

FMIN (vector): Floating-point minimum (vector).

FMINNM (scalar): Floating-point Minimum Number (scalar).

FMINNM (vector): Floating-point Minimum Number (vector).

FMINNMP (scalar): Floating-point Minimum Number of Pair of elements (scalar).

FMINNMP (vector): Floating-point Minimum Number Pairwise (vector).

FMINMV: Floating-point Minimum Number 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.

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).

FMLS (by element): Floating-point fused Multiply-Subtract from accumulator (by element).

FMLS (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).

FMSUB: Floating-point Negated fused Multiply-Add (scalar).

FMUL (by element): Floating-point Multiply (by element).

FMUL (scalar): Floating-point Multiply (scalar).

FMUL (vector): Floating-point Multiply (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).

FNMADD: Floating-point Negated fused Multiply-Add (scalar).

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 (scalar): 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).
FRINZ (scalar): Floating-point Round to Integral, toward Zero (scalar).
FRINZ (vector): Floating-point Round to Integral, toward Zero (vector).
FRSORTE: Floating-point Reciprocal Square Root Estimate.
FRSQRTE: 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).
REVI (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).
SADLPL: Signed Add Long Pairwise.
SADLVM: 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).

SQDMLUL (by element): Signed saturating Doubling Multiply returning High half (by element).

SQDMULH (vector): Signed saturating Doubling Multiply returning High half.

SQDMULL, SQDMLSL2 (by element): Signed saturating Doubling Multiply Long (by element).


SQNEG: Signed saturating Negate.

SORDMLAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

SORDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

SORDMLSH (by element): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

SORDMLSH (vector): Signed saturating Rounding Doubling Multiply Subtract returning High half (vector).

SORDMULH (by element): Signed saturating Rounding Doubling Multiply returning High half (by element).

SORDMULH (vector): Signed saturating Rounding Doubling Multiply returning High half.

SQRSHL: Signed saturating Rounding Shift Left (register).

SORSHRN, SORSHRN2: Signed saturating Rounded Shift Right Narrow (immediate).

SORSHRUN, SORSHRUN2: Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

SORSHL (immediate): Signed saturating Shift Left (immediate).

SORSHL (register): Signed saturating Shift Left (register).

SORSHL: Signed saturating Shift Left Unsigned (immediate).

SORSHRN, SORSHRN2: Signed saturating Shift Right Narrow (immediate).

SORSHRUN, SORSHRUN2: Signed saturating Shift Right Unsigned Narrow (immediate).

SQRSH: Signed saturating Subtract.

SQRXTN, SQRTN2: Signed saturating extract Narrow.

SQRXTN, SQRTN2: 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).

SSHTL: Signed Shift Left (register).

SSHLL, SSHLL2: Signed Shift Left Long (immediate).

SSHRT: Signed Shift Right (immediate).

SSRAL: 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.

**SXTL, SXTL2**: Signed extend Long: an alias of SSHLL, SSHLL2.

**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.

**UCVT (scalar, fixed-point)**: Unsigned fixed-point Convert to Floating-point (scalar).

**UCVT (scalar, integer)**: Unsigned integer Convert to Floating-point (scalar).

**UCVT (vector, fixed-point)**: Unsigned fixed-point Convert to Floating-point (vector).

**UCVT (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).

URSRL: Unsigned Rounding Shift Right (immediate).

URSRL2: Unsigned Rounding Shift Right and Accumulate (immediate).

USRSTA: 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>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>0 1 0 1 1</td>
<td>U</td>
</tr>
</tbody>
</table>

Scalar

ABS <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>Rd</th>
<th>Rn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Q 0 1 1 1 1 0</td>
<td>1 0 0 0 0</td>
<td>0 1 0 1 1</td>
<td>U</td>
</tr>
</tbody>
</table>

Vector

ABS <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”:
|---|---|
| size | <V>
| 0x | RESERVED
| 10 | RESERVED
| 11 | 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> | 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 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

| 31 30 29 28 27 26 25 24 23 22 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 0 0 0 1 | Rn | Rd |
| U               |                   |

Scalar

ADD <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 0 0 1 1 1 1 0 | size | 1 | Rm | 1 0 0 0 0 1 | Rn | Rd |
| U               |                   |

Vector

ADD <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>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;
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.

Three registers, not all the same type

ADDHN(2) <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<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":

<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

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 | Rd
| Rn |

Advanced SIMD

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&lt; 0x 10 11</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&lt; 0x 10 11</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.

<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>

**Three registers of the same type**

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.

![Register Format]

### Advanced SIMD

**ADDV <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;

### Assembler Symbols

<**V**>  Is the destination width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;<strong>V</strong>&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;<strong>T</strong>&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.

Advanced SIMD

AESD \texttt{<Vd>.16B, <Vn>.16B}

integer d = \texttt{UINT(Rd)};
integer n = \texttt{UINT(Rn)};
if \texttt{!HaveAESExt()} then UNDEFINED;

Assembler Symbols

\texttt{<Vd>} Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
\texttt{<Vn>} Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

\texttt{AArch64.CheckFPAdvSIMDEnabled();}

\texttt{bits(128) operand1 = V[d];}
\texttt{bits(128) operand2 = V[n];}
\texttt{bits(128) result;}
\texttt{result = operand1 \texttt{EOR} operand2;}
\texttt{result = AESInvSubBytes(AESInvShiftRows(result));}
\texttt{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.

Advanced SIMD

AESE \(<Vd>.16B, <Vn>.16B\)

integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESEExt() 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 = 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.
AESIMC

AES inverse mix columns.

Advanced SIMD

AESIMC $<Vd>$.16B, $<Vn>$.16B

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
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.

Advanced SIMD

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 `CPACR_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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Rn</td>
<td>1</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>
<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>
</tbody>
</table>

Three registers of the same type

AND <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

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
result = operand1 AND operand2;
V[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 registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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 ARMv8.2-SHA 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  | 0  | 1  |    |   | Rm |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Rn | Rd |

Advanced SIMD

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.
BFCVT

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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Rn  Rd

Single-precision to BFloat16

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];
bits(16) result;
result = FPConvertBF(operand, FPCR);
V[d] = result;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 and BFloat16
(Armv8.6)

<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>Rn</td>
</tr>
</tbody>
</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>
<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 &lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) operand = V[n];
bits(64) result;
for e = 0 to elements-1
    Elem[result, e, 16] = FPConvertBF(Elem[operand, e, 32], FPCR);
Vpart[d, part] = result;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 1  | L  | M  | Rm | 1  | 1  | 1  | 1  | H  | 0  | Rn | Rd |

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

CheckFPAvSIMDEnabled64();
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)**

```
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 | 0 | 0 | 1 | 0 | Rm | 1 | 1 | 1 | 1 | 1 | Rn | Rd |
```

**Operation**

```
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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 1  | 1  | 1  | 0  | L  | M  | Rm | 1  | 0  | 1  | 0  | 1  | 1  | 0  | H  | 0  | Rd |

Vector

BFMLAL <bt> <Vd>.4S, <Vn>.8H, <Vm>.H[<index>]

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

<bt> 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>

<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, in the range V0 to V15, encoded in the "Rm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

Operation

CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) operand3 = V[d];
bits(128) result;

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] = FPMulAdd(addend, element1, element2, FPCR);
V[d] = result;
BFMLALB, BFMLALT (vector)

 BFMLALB, BFMLALT (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 this instruction is supported.

**Vector**
(Armv8.6)

<table>
<thead>
<tr>
<th>Q</th>
<th>Rm</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1 1 1 0 1 1 0</td>
<td>1 1 1 1 1</td>
</tr>
</tbody>
</table>

**BFMLAL<bt>**<Vd>.4S, <Vn>.8H, <Vm>.8H

if !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**

<bt> 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>

<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**

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

bits(32) element1 = Elem[operand1, 2*e+sel, 16]:Zeros(16);
bits(32) element2 = Elem[operand2, 2*e+sel, 16]:Zeros(16);
bits(32) addend = Elem[operand3, e, 32];
Elem[result, e, 32] = FPMulAdd(addend, element1, element2, FPCR);

V[d] = result;
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)

<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>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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Rm | 1  | 1  | 1  | 0  | 1  | 1  |
Rn | 1  | 1  | 0  | 1  | 1  |
Rd | 0  | 0  |

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.

16-bit (cmode == 10x1)

BIC <Vd>.<T>, #<imm8>{, LSL <amount>}

32-bit (cmode == 0xx1)

BIC <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 '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 = 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>&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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | Rm | 0  | 0  | 0  | 1  | 1  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Three registers of the same type**

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;

**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 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;
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.

![32-bit register](image)

Three registers of the same type

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

```plaintext
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.

Three registers of the same type

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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 | 1 | 1 | Rm | 0 | 0 | 0 | 1 | 1 | Rd |

Three registers of the same type

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 **CPACR_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 SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
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.
**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.

```
31 30 29 28 27 26 25 24 23 22 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 | size 1 0 0 0 0 0 0 1 0 0 1 0 | Rd
U 0 Q Rn
```

**Vector**

CLZ `<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;
```

**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;

total elements = datasize / esize;
for e = 0 to elements - 1
  count = if countop == CountOp_CLS then
           CountLeadingSignBits(Elem[operand, e, esize])
          else
           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.

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 1  | 0  | size | 1  | Rm  | 1  | 0  | 0  | 0  | 1  | 1  | Rn  | 1  | 0  | 0  | 1  | 1  | Rd |

Scalar

CMEQ <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 | 1  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm  | 1  | 0  | 0  | 0  | 1  | 1  | Rn  | 1  | 0  | 0  | 1  | 1  | Rd |

Vector

CMEQ <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>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(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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

31 30 29 28 27 26 25 24 23 22 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 Rd Rn

Scalar

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;

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 1 1 1 1 0 size 1 0 0 0 0 0 1 0 0 1 1 0 Rd Rn

Vector

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;

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

<\text{V}> \quad \text{Is a width specifier, encoded in “size”:}

<table>
<thead>
<tr>
<th>size</th>
<th>\text{&lt;\text{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>

<\text{d}> \quad \text{Is the number of the SIMD&FP destination register, encoded in the “Rd” field.}

<\text{n}> \quad \text{Is the number of the SIMD&FP source register, encoded in the “Rn” field.}

<\text{Vd}> \quad \text{Is the name of the SIMD&FP destination register, encoded in the “Rd” field.}

<\text{T}> \quad \text{Is an arrangement specifier, encoded in “size:Q”:}

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>\text{&lt;\text{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{Vn}> \quad \text{Is the name of the SIMD&FP source register, encoded in the “Rn” field.}

Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \text{V}[n];
bits(datasize) result;
integer element;
boolean test_passed;
for e = 0 to elements-1
    element = \text{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 \text{Ones()} else \text{Zeros();}
\end{verbatim}

\text{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 |
|----------------------|----------------------|----------------------|----------------------|
| size                | Rm                   | Rn                   | Rd                   |
| 0 1 0 1 1 1 1 0     | 0 0 1 1 1 1          | eq                   |
```

Scalar

```
CMGE <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

```
| 31 30 29 28 27 26 25 24 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                | Rm                   | Rn                   | Rd                   |
| 0 Q 0 1 1 1 1 0     | 0 0 1 1 1 1          | eq                   |
```

Vector

```
CMGE <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>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**

```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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \texttt{CPACR\_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: \texttt{Scalar} and \texttt{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 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | Rn | Rd |
U     op
```

**CMGE** \texttt{<V><d>, <V><n>}, \#0

integer \texttt{d} = \texttt{UInt}(Rd);
integer \texttt{n} = \texttt{UInt}(Rn);

if \texttt{size} \neq '11' then UNDEFINED;
integer \texttt{esize} = 8 \times \texttt{UInt}(\text{size});
integer \texttt{datasize} = \text{esize};
integer \texttt{elements} = 1;

\texttt{CompareOp} \ comparison;
case \texttt{op}:U of
  when '00' \ comparison = \texttt{CompareOp\_GT};
  when '01' \ comparison = \texttt{CompareOp\_GE};
  when '10' \ comparison = \texttt{CompareOp\_EQ};
  when '11' \ comparison = \texttt{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 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | Rn | Rd |
U     op
```

**CMGE** \texttt{<Vd>.<T>, <Vn>.<T>}, \#0

integer \texttt{d} = \texttt{UInt}(Rd);
integer \texttt{n} = \texttt{UInt}(Rn);

if \texttt{size}:Q \neq '110' then UNDEFINED;
integer \texttt{esize} = 8 \times \texttt{UInt}(\text{size});
integer \texttt{datasize} = if \texttt{Q} \neq '1' then 128 else 64;
integer \texttt{elements} = \text{datasize} \div \text{esize};

\texttt{CompareOp} \ comparison;
case \texttt{op}:U of
  when '00' \ comparison = \texttt{CompareOp\_GT};
  when '01' \ comparison = \texttt{CompareOp\_GE};
  when '10' \ comparison = \texttt{CompareOp\_EQ};
  when '11' \ comparison = \texttt{CompareOp\_LE};
Assembler Symbols

<\text{V}> \quad \text{Is a width specifier, encoded in “size”:}
\begin{tabular}{c|c}
\text{\texttt{size}} & <\text{V}> \\
0x & RESERVED \\
10 & RESERVED \\
11 & D \\
\end{tabular}

<\text{d}> \quad \text{Is the number of the SIMD\&FP destination register, encoded in the “Rd” field.}

<\text{n}> \quad \text{Is the number of the SIMD\&FP source register, encoded in the “Rn” field.}

<\text{Vd}> \quad \text{Is the name of the SIMD\&FP destination register, encoded in the “Rd” field.}

<\text{T}> \quad \text{Is an arrangement specifier, encoded in “size:Q”:}
\begin{tabular}{c|c|c}
\text{\texttt{size}} & \text{Q} & <\text{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 \\
\end{tabular}

<\text{Vn}> \quad \text{Is the name of the SIMD\&FP source register, encoded in the “Rn” field.}

Operation

\texttt{CheckFPAdvSIMDEnabled64}();
\text{bits(datasize) operand = } \texttt{V}[\text{\texttt{n}}];
\text{bits(datasize) result};
\text{integer element};
\text{boolean test\_passed};
\text{for e = 0 to elements-1}
\quad \text{element = } \texttt{SInt(Elem[operand, e, esize])};
\quad \text{case comparison of}
\quad \text{when } \texttt{CompareOp\_GT} \text{ test\_passed = element > 0};
\quad \text{when } \texttt{CompareOp\_GE} \text{ test\_passed = element } \geq 0;
\quad \text{when } \texttt{CompareOp\_EQ} \text{ test\_passed = element } = 0;
\quad \text{when } \texttt{CompareOp\_LE} \text{ test\_passed = element } \leq 0;
\quad \text{when } \texttt{CompareOp\_LT} \text{ test\_passed = element } < 0;
\quad \texttt{Elem[result, e, esize] = if test\_passed then } \texttt{Ones()} \text{ else } \texttt{Zeros()};
\text{\texttt{V}[d]} = \texttt{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}
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

| 31 30 29 28 27 26 25 24 23 22 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 0 1 1 0 1 | Rn | Rd | u | eq |

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 | Rm | 0 0 1 1 0 1 | Rn | Rd | u | eq |

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>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**

```pseudocode
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 \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{Scalar} and \texttt{Vector}

\textbf{Scalar}

\begin{verbatim}
 Scalar 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;
\end{verbatim}

\textbf{Vector}

\begin{verbatim}
 Vector 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;
\end{verbatim}
**Assembler Symbols**

<i>V</i> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;i&gt;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>

<i>d</i> Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

<i>n</i> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<i>Vd</i> Is the name of the SIMD&FP destination register, encoded in the “Rd” field.

<i>T</i> Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;i&gt;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>

<i>Vn</i> 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

```
| 31 30 29 28 27 26 25 24 23 22 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               | 0               | 1               | 1               | 0               | 1               | Rd              | eq              |
| U               | unsigned        |
```

### 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               | 0               | size            | 1               | Rm              | 0               | 0               | 1               | 1               | 0               | 1               | Rd              | eq              |
| Q               | unsigned        |
```

### Assembler Symbols

- `<V>` is a width specifier, encoded in “size”:
  - **size**
    | 0x  | RESERVED |
    | 10  | RESERVED |
    | 11  | 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> 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];
bits(datasize) result;
integer element1;
ineger 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 1 0 | size | 1 | Rm | 0 0 1 1 1 1 | Rn | Rd |
| U | eq |

**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 0 1 1 1 1 | Rn | Rd |
| U | eq |

**Assembler Symbols**

- `<V>` Is a width specifier, encoded in “size”:
  - size  `<V>`
    - 0x: RESERVED
    - 10: RESERVED
    - 11: D

- `<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>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>

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 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.
**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**

```
[01111111101000000101110] Rn    Ud
```

```
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**

```
[01111111101000000101110] Rn    Ud
```

```
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

<\text{V}> \quad \text{Is a width specifier, encoded in “size”:}

\begin{array}{|c|c|}
\hline
\text{size} & <\text{V}> \\
\hline
0 & \text{RESERVED} \\
10 & \text{RESERVED} \\
11 & D \\
\hline
\end{array}

<\text{d}> \quad \text{Is the number of the SIMD\&FP destination register, encoded in the “Rd” field.}

<\text{n}> \quad \text{Is the number of the SIMD\&FP source register, encoded in the “Rn” field.}

<\text{Vd}> \quad \text{Is the name of the SIMD\&FP destination register, encoded in the “Rd” field.}

<\text{T}> \quad \text{Is an arrangement specifier, encoded in “size:Q”:}

\begin{array}{|c|c|}
\hline
\text{size} & Q & <\text{T}> \\
\hline
00 & 0 & \text{8B} \\
00 & 1 & \text{16B} \\
01 & 0 & \text{4H} \\
01 & 1 & \text{8H} \\
10 & 0 & \text{2S} \\
10 & 1 & \text{4S} \\
11 & 0 & \text{RESERVED} \\
11 & 1 & \text{2D} \\
\hline
\end{array}

<\text{Vn}> \quad \text{Is the name of the SIMD\&FP source register, encoded in the “Rn” field.}

Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \text{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();
\end{verbatim}

\text{V}[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.
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

Scalar CMLT &d, &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

Vector CMLT &d,.T, &n,.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>0x</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;V&gt;</td>
<td></td>
<td></td>
<td></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;

type 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.
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 \texttt{CPACR	extunderscore EL1}, \texttt{CPTR	extunderscore EL2}, and \texttt{CPTR	extunderscore 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{Scalar} and \texttt{Vector}

\textbf{Scalar}

\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 O 1 1 1 1 0 | size | Rm | 1 0 0 0 1 1 | Rn | Rd
\end{verbatim}

\textbf{Vector}

\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 Q 0 1 1 1 0 | size | Rm | 1 0 0 0 1 1 | Rn | Rd
\end{verbatim}

\textbf{Assembler Symbols}

\begin{verbatim}
<V> Is a width specifier, encoded in "size":
  size  <V>
  0x  RESERVED
  10  RESERVED
  11  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.
\end{verbatim}
<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;
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 1  | 0  | 1  | 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>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**

```plaintext
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;
```

```
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

\[
\begin{array}{cccccccccccccccccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & Rn & Rd \\
\end{array}
\]

Scalar

\[
\text{DUP }<V><d>, <Vn>.<T>[<index>]}
\]

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

\[
\begin{array}{cccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & Rn & Rd \\
\end{array}
\]

Vector

\[
\text{DUP }<Vd>.<T>, <Vn>.<Ts>[<index>]}
\]

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

\(<T>\)

For the scalar variant: is the element width specifier, encoded in “imm5”:
For the vector variant: is an arrangement specifier, encoded in “imm5:Q”:

<table>
<thead>
<tr>
<th>imm5</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0000</td>
<td>x</td>
</tr>
<tr>
<td>xxxx1</td>
<td>0</td>
</tr>
<tr>
<td>xxxx2</td>
<td>1</td>
</tr>
<tr>
<td>xxx10</td>
<td>0</td>
</tr>
<tr>
<td>xxx10</td>
<td>1</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
</tr>
<tr>
<td>x1000</td>
<td>1</td>
</tr>
</tbody>
</table>

<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>x</td>
</tr>
<tr>
<td>xxxx1</td>
<td>0</td>
</tr>
<tr>
<td>xxxx2</td>
<td>1</td>
</tr>
<tr>
<td>xxx10</td>
<td>0</td>
</tr>
<tr>
<td>xxx10</td>
<td>1</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
</tr>
<tr>
<td>x1000</td>
<td>1</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>x</td>
</tr>
<tr>
<td>xxxx1</td>
<td>B</td>
</tr>
<tr>
<td>xxxx2</td>
<td>H</td>
</tr>
<tr>
<td>xxx10</td>
<td>S</td>
</tr>
<tr>
<td>xxx10</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>x</td>
</tr>
<tr>
<td>xxxx1</td>
<td>imm5&lt;4:1&gt;</td>
</tr>
<tr>
<td>xxxx2</td>
<td>imm5&lt;4:2&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:3&gt;</td>
</tr>
<tr>
<td>xxx10</td>
<td>imm5&lt;4:4&gt;</td>
</tr>
<tr>
<td>xx100</td>
<td>0</td>
</tr>
<tr>
<td>xx100</td>
<td>1</td>
</tr>
<tr>
<td>x1000</td>
<td>0</td>
</tr>
<tr>
<td>x1000</td>
<td>1</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.

```
|   31   30  29 28 27 26 25 24 23 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 | 0 | 0 | 0 | 0 | imm5 | 0 | 0 | 0 | 1 | 1 | Rn | Rd |   |
```

Advanced SIMD

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;
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 “imm5:Q”:

```
<imm5> | <Q> | <T> |
--------|-----|-----|
  0000  | x   | RESERVED |
  xxx1  | 0   | 8B     |
  xxx1  | 1   | 16B    |
  xxx10 | 0   | 4H     |
  xxx10 | 1   | 8H     |
  xx100 | 0   | 2S     |
  xx100 | 1   | 4S     |
  x1000 | 0   | RESERVED |
  x1000 | 1   | 2D     |
```

`<R>` Is the width specifier for the general-purpose source register, encoded in “imm5”:

```
<imm5> | <R> |
--------|-----|
  0000  | RESERVED |
  xxx1  | W       |
  xxx10 | W       |
  xx100 | W       |
  x1000 | X       |
```

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 ARMv8.2-SHA is implemented.

Advanced SIMD
(Armv8.2)

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | Rm |   | Ra |   | Rn |   | Rd |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

Advanced SIMD

```
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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Three registers of the same type

EOR <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) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n];

operand1 = V[m];
operand2 = Zeros();
operand3 = Ones();
V[d] = operand1 EOR ((operand2 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.
EXT

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.

Advanced SIMD

EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if Q == '0' && mm4<3> == '1' then UNDEFINED;

integer datasize = if Q == '1' then 128 else 64;
integer position = UInt(imm4) << 3;

Asmblered 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.
<index> Is the lowest numbered byte element to be extracted, encoded in "Q:imm4":

<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>

Operation

CheckFPAdvSIMDEnabled64();
bvnt(datasize) hi = V[m];
bvnt(datasize) lo = V[n];
bvnt(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   | 0   | 1   | 1   | 0   | Rm  | 0   | 0   | 1   | 0   | 1   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Scalar half precision

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   | 0   | 1   | 1   | 0   | sz  | 1   | Rm  | 1   | 1   | 0   | 1   | 0   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Scalar single-precision and double-precision

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   | 0   | 1   | 1   | 1   | 0   | 1   | 1   | 0   | Rm  | 0   | 0   | 0   | 1   | 0   | 1   | Rn  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Vector half precision

FABD \( <Vd>., <T>, <Vn>.<T>, <Vm>.<T> \)

if !\texttt{HaveFP16Ext}() then UNDEFINED;

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{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

FABD \( <Vd>, <T>, <Vn>, <T>, <Vm> <T> \)

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
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 abs = (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.
\(<Hm>\) 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 vector 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 vector 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;
bits(esize) diff;

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)*

| 31 30 29 28 27 26 25 24 23 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 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | Rn | Rd |
| U |

Half-precision

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**

| 31 30 29 28 27 26 25 24 23 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 | 1 | 1 | 1 | 0 | Rn | Rd |
| U |

Single-precision and double-precision

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”:

<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>

<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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 CPACR_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)

FABS <Hd>, <Hn>

**Single-precision (ftype == 00)**

FABS <Sd>, <Sn>

**Double-precision (ftype == 01)**

FABS <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

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;

**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();

bits(datasize) result;
bits(datasize) operand = V[n];
result = FPAbs(operand);
V[d] = 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)

```
|    | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 1 | 0 | 1 | 1 | Rn | Rd |
|----|---|---|---|---|---|---|---|---|---|---|---|----|---|---|---|---|---|---|---|---|---|
| U  | E | E |   |   |   |   |   |   |   |   |   | ac |
```

Scalar half precision

FACGE <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

```
|    | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 1 | 1 | Rn | Rd |
|----|---|---|---|---|---|---|---|---|---|----|---|----|---|---|---|---|---|---|---|---|
| U  | E | E |   |   |   |   |   |   |   |   |   | ac |
```
Scalar single-precision and double-precision

FACGE <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;

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 0  | 1  | 1  | Rn | 0  | 0  | 1  | 0  | 1  | 1  | Rd |
| U  | E  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Vector half precision

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 1  | 1  | Rn | 1  | 1  | 1  | 0  | 1  | 1  | Rd |
| U  | E  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

FACGE
Vector single-precision and double-precision

FACGE <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;
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>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</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>      For the vector 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 vector 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

`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 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 | Rd |
| U | E | ac |

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 | Rd |
| U | E | ac |

Scalar half precision

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

FACGT
Scalar single-precision and double-precision

\[ \text{FACGT} \langle V \rangle<d>, \langle V \rangle<n>, \langle V \rangle<m> \]

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;
\text{CompareOp} cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = \text{CompareOp_EQ}; abs = FALSE;
  when '010' cmp = \text{CompareOp_GE}; abs = FALSE;
  when '011' cmp = \text{CompareOp_GE}; abs = TRUE;
  when '110' cmp = \text{CompareOp_GT}; abs = FALSE;
  when '111' cmp = \text{CompareOp_GT}; abs = TRUE;
otherwise UNDEFINED;

Vector half precision

(\text{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  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | Rm | 0  | 0  | 1  | 0  | 1  | 1  | Rn | Rd |

Vector half precision

\[ \text{FACGT} \langle Vd \rangle.<T>, \langle Vn \rangle.<T>, \langle Vm \rangle.<T> \]

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;
\text{CompareOp} cmp;
boolean abs;

case E:U:ac of
  when '000' cmp = \text{CompareOp_EQ}; abs = FALSE;
  when '010' cmp = \text{CompareOp_GE}; abs = FALSE;
  when '011' cmp = \text{CompareOp_GE}; abs = TRUE;
  when '110' cmp = \text{CompareOp_GT}; abs = FALSE;
  when '111' cmp = \text{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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 1  | 1  | Rn | Rd |

\text{FACGT}
Vector single-precision and double-precision

\[ \text{FACGT} \ <V_d>.<T>, \ <V_n>.<T>, \ <V_m>.<T> \]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(R_d); \\
\text{integer } n &= \text{UInt}(R_n); \\
\text{integer } m &= \text{UInt}(R_m); \\
\text{if } sz:Q = '10' \text{ then UNDEFINED;} \\
\text{integer } esize &= 32 \ll \text{UInt}(sz); \\
\text{integer } datasize &= \text{if } Q = '1' \text{ then 128 else 64}; \\
\text{integer } elements &= \text{datasize DIV esize}; \\
\text{CompareOp } cmp; \\
\text{boolean } abs;
\end{align*}
\]

\[
\text{case E:U:ac of}
\]
\[
\begin{align*}
\text{when } '000' & \text{ cmp } = \text{CompareOp_EQ}; \text{ abs } = \text{FALSE}; \\
\text{when } '010' & \text{ cmp } = \text{CompareOp_GE}; \text{ abs } = \text{FALSE}; \\
\text{when } '011' & \text{ cmp } = \text{CompareOp_LE; } \text{ abs } = \text{TRUE}; \\
\text{when } '110' & \text{ cmp } = \text{CompareOp_GT; abs } = \text{FALSE}; \\
\text{when } '111' & \text{ cmp } = \text{CompareOp_GT; abs } = \text{TRUE}; \\
\text{otherwise UNDEFINED;}
\end{align*}
\]

Assembler Symbols

\[
\begin{align*}
<H_d> & \quad \text{Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.} \\
<H_n> & \quad \text{Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.} \\
<H_m> & \quad \text{Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.} \\
<V> & \quad \text{Is a width specifier, encoded in \"sz\":}
\end{align*}
\]

\[
\begin{array}{c|c}
\text{sz} & <V> \\
\hline
0 & 5 \\
1 & 0 \\
\end{array}
\]

\[
<d> & \quad \text{Is the number of the SIMD&FP destination register, in the \"Rd\" field.} \\
<n> & \quad \text{Is the number of the first SIMD&FP source register, encoded in the \"Rn\" field.} \\
<m> & \quad \text{Is the number of the second SIMD&FP source register, encoded in the \"Rm\" field.} \\
<V_d> & \quad \text{Is the name of the SIMD&FP destination register, encoded in the \"Rd\" field.} \\
<T> & \quad \text{For the vector half precision variant: is an arrangement specifier, encoded in \"Q\":}
\end{align*}
\]

\[
\begin{array}{c|c}
\text{Q} & <T> \\
\hline
0 & 4H \\
1 & 8H \\
\end{array}
\]

\[
\text{For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in \"sz:Q\":}
\]

\[
\begin{array}{c|cc}
\text{sz} & \text{Q} & <T> \\
\hline
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & \text{RESERVED} \\
1 & 1 & 2D \\
\end{array}
\]

\[
<V_n> & \quad \text{Is the name of the first SIMD&FP source register, encoded in the \"Rn\" field.} \\
<V_m> & \quad \text{Is the name of the second SIMD&FP source register, encoded in the \"Rm\" field.}
\]
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(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 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;
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   |  1   |  0   |  0   |  1   |  0   |  0   |  0   |  1   |  0   |  1   |  1   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  1   |  0   |  1   |  0   |  1   |  1   |  0   |  1   |  0   |  1   |  Rm  |
| U     |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
```

Half-precision

FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveFP16Ext() then UNDEFINED;

ingenient d = UInt(Rd);
genient n = UInt(Rn);
genient m = UInt(Rm);
genient esize = 16;
genient datasize = if Q == '1' then 128 else 64;
genient 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   |  1   |  0   |  0   |  1   |  0   |  0   |  1   |  0   |  1   |  1   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  1   |  0   |  1   |  0   |  1   |  1   |  0   |  1   |  0   |  1   |  0   |  1   |  Rm  |
| U     |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |      |
```

Single-precision and double-precision

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";

FADD (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>

<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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

$$\begin{array}{cccccccccccccc}
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & ftype & 1 & Rm & 0 & 0 & 1 & 0 & 1 & 0 & Rn & Rd \\
\end{array}$$

op

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 datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 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(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
result = FPAdd(operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

| 31 30 29 28 27 26 25 24 23 22 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 1 1 0 1 1 0 0 0 0 0 1 1 0 1 1 0 | Rn | Rd |

**Half-precision**

FADDP <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 0 0 0 1 1 0 1 1 0 0 0 0 0 1 1 0 1 1 0 | Rn | Rd |

**Single-precision and double-precision**

FADDP <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":

| 0 | H |
| 1 | RESERVED |

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

| 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":
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_FADD, operand, esize);
```

---

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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

(Asmv8.2)

| 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  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  |    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

**Half-precision**

FADDP <Vd>.<T>, <Vm>.<T>, <Vn>.<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  | 0  | 0  | 1  | 0  | Rm | 1  | 1  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  |    |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

**Single-precision and double-precision**

FADDP <Vd>.<T>, <Vm>.<T>, <Vn>.<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”:
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];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 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 current Security state and
Exception level, an attempt to execute the instruction might be trapped.

Vector
(Armv8.3)

<table>
<thead>
<tr>
<th></th>
<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>Rd</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

Vector

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 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.
It raises an Invalid Operation exception only if either operand is a signaling 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 |    | 0  |    | nzcv | op |

**Half-precision (ftype == 11)**
(Armv8.2)

FCCMP <Hn>, <Hm>, #<nzcv>, <cond>

**Single-precision (ftype == 00)**

FCCMP <Sn>, <Sm>, #<nzcv>, <cond>

**Double-precision (ftype == 01)**

FCCMP <Dn>, <Dm>, #<nzcv>, <cond>

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.

NaNs
The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the **FPSCR** flags being set to N=0, Z=0, C=1, and V=1.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;

operand2 = V[m];

if ConditionHolds(cond) then
  flags = FPCompare(operand1, operand2, FALSE, FPCR);
PSTATE.<N,Z,C,V> = flags;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

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.

\[
\begin{array}{cccccccccccccc}
0 & 0 & 0 & 1 & 1 & 1 & 0 & ftype & 1 & Rm & cond & 0 & 1 & Rn & 1 & nzcv & \hline
\end{array}
\]

**Half-precision (ftype == 11)**  
(Armv8.2)

\[
\text{FCCMPE } <Hn>, <Hm>, #<nzcv>, <cond>
\]

**Single-precision (ftype == 00)**

\[
\text{FCCMPE } <Sn>, <Sm>, #<nzcv>, <cond>
\]

**Double-precision (ftype == 01)**

\[
\text{FCCMPE } <Dn>, <Dm>, #<nzcv>, <cond>
\]

integer \( n = \text{UInt}(Rn); \)
integer \( m = \text{UInt}(Rm); \)

integer datasize;

\text{case ftype of}
  \text{when '00' datasize = 32;}
  \text{when '01' datasize = 64;}
  \text{when '10' UNDEFINED;}
  \text{when '11'
    if HaveFP16Ext() then}
    \text{datasize = 16;}
  \text{else}
    \text{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.
NaNs
The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.
FCCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = V[n];
bits(datasize) operand2;
operand2 = V[m];
if ConditionHolds(cond) then
  flags = FPCompare(operand1, operand2, TRUE, FPCR);
PSTATE.<N,Z,C,V> = flags;
FCMEQ (register)

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)

```
31 30 29 28 27 26 25 24 23 22 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 1 0 0 1 | Rn | Rd
U | E | ac
```

Scalar half precision

FCMEQ <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 0 1 1 1 1 0 0 | sz | Rm | 1 1 1 0 0 1 | Rn | Rd
U | E | ac
```
Scalar single-precision and double-precision

FCMEQ \(<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;
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)

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</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>
</tr>
<tr>
<td>Rm</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>
</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>
</tr>
</tbody>
</table>

Vector half precision

FCMEQ \(<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

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>sz</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd</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>0</td>
</tr>
<tr>
<td>Rm</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>0</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>
</tr>
</tbody>
</table>
Vector single-precision and double-precision

FCMEQ <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;

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 vector 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 vector 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;
boolean test_passed;

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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 0 1 1 1 1 0 1 1 1 1 0 0 0 1 1 0 1 1 0 | Rn | Rd
U op

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 0 1 sz 1 0 0 0 0 0 1 1 0 | Rn | Rd
U op
Scalar single-precision and double-precision

FCMEQ \( \langle V \rangle \langle d \rangle, \langle V \rangle \langle n \rangle \), \#0.0

integer \( d = \text{UInt}(\text{Rd}) \);
integer \( n = \text{UInt}(\text{Rn}) \);

integer esize = 32 \( \ll \) \( \text{UInt}(\text{sz}) \);
integer datasize = esize;
integer elements = 1;

\text{CompareOp} \ comparison;
case op:U of
  when '00' comparison = \text{CompareOp_GT};
  when '01' comparison = \text{CompareOp_GE};
  when '10' comparison = \text{CompareOp_EQ};
  when '11' comparison = \text{CompareOp_LE};

Vector half precision

(\text{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  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |

<table>
<thead>
<tr>
<th>U</th>
<th>o p</th>
</tr>
</thead>
</table>

Vector half precision

FCMEQ \( \langle Vd \rangle \langle .T \rangle, \langle Vn \rangle \langle .T \rangle \), \#0.0

if !\text{HaveFP16Ext}() then UNDEFINED;

integer \( d = \text{UInt}(\text{Rd}) \);
integer \( n = \text{UInt}(\text{Rn}) \);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

\text{CompareOp} \ comparison;
case op:U of
  when '00' comparison = \text{CompareOp_GT};
  when '01' comparison = \text{CompareOp_GE};
  when '10' comparison = \text{CompareOp_EQ};
  when '11' 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rn | Rd |

<table>
<thead>
<tr>
<th>U</th>
<th>o p</th>
</tr>
</thead>
</table>
Vector single-precision and double-precision

FCMEQ $<Vd>$. $<T>$, $<Vn>$. $<T>$, #0.0

integer $d = \text{UInt}(\text{Rd})$;
integer $n = \text{UInt}(\text{Rn})$;

if $\text{sz}:\text{Q} == '10'$ then UNDEFINED;
integer esize = 32 \ll \text{UInt}(\text{sz})
integer datasize = if $\text{Q} == '1'$ then 128 else 64;
integer elements = datasize DIV esize;

\textbf{CompareOp} comparison;
\textbf{case} op:U of
  when '00' comparison = \textbf{CompareOp GT};
  when '01' comparison = \textbf{CompareOp GE};
  when '10' comparison = \textbf{CompareOp EQ};
  when '11' comparison = \textbf{CompareOp LE};

\textbf{Assembler Symbols} \\

$<\text{Hd}>$ 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.
$<\text{V}>$ Is a width specifier, encoded in "sz":

\begin{tabular}{c|c}
  \textbf{sz} & \textbf{<V>} \\
  \hline
  0 & S \\
  1 & D \\
\end{tabular}

$d$ 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.

$<\text{Vd}>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$<\text{T}>$ For the vector half precision variant: is an arrangement specifier, encoded in "Q":

\begin{tabular}{c|c}
  \textbf{Q} & \textbf{<T>} \\
  \hline
  0 & 4H \\
  1 & 8H \\
\end{tabular}

For the vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\begin{tabular}{c|c|c}
  \textbf{sz} & \textbf{Q} & \textbf{<T>} \\
  \hline
  0 & 0 & 2S \\
  0 & 1 & 4S \\
  1 & 0 & \text{RESERVED} \\
  1 & 1 & 2D \\
\end{tabular}

$<\text{Vn}>$ 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) 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;
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 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

Scalar half precision

FCMGE <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

Scalar single-precision and double-precision
Scalar single-precision and double-precision

FCMGE $<V><d>$, $<V><n>$, $<V><m>$

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

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)

Vector single-precision and double-precision

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

FCMGE <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;
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>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, 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 vector 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 vector 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

```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 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;

---

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

31 30 29 28 27 26 25 24 23 22 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 1 0 0 0 1 1 0 0 1 1 0 0 0 1 1 0 0 0 1 1 0 0
 U   op

Scalar half precision

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

31 30 29 28 27 26 25 24 23 22 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 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0
 U   op

Scalar single-precision and double-precision
Scalar single-precision and double-precision

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 |
|-----------------|-----------------|-----------------|
|       Q         |       1         |       0         |
|       1         |       1         |       1         |
|       1         |       1         |       1         |
|       0         |       0         |       1         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |
|       1         |       1         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |

Vector half precision

FCMGE \(<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 |
|-----------------|-----------------|-----------------|
|       Q         |       1         |       0         |
|       1         |       1         |       1         |
|       1         |       0         |       0         |
|       0         |       0         |       0         |
|       0         |       1         |       1         |
|       0         |       0         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |
|       0         |       1         |       0         |

FCMGE (zero)
Vector single-precision and double-precision

FCMGE $<Vd>$, $<T>$, $<Vn>$, $<T>$, #0.0

integer $d = \text{UInt}(\text{Rd})$;
integer $n = \text{UInt}(\text{Rn})$;

if $\text{sz:Q} == '10'$ then UNDEFINED;
integer $\text{esize} = 32 << \text{UInt}(\text{sz})$;
integer $\text{datasize} = \text{if} \ Q == '1' \ \text{then} \ 128 \ \text{else} \ 64$;
integer $\text{elements} = \text{datasize} \ \text{DIV} \ \text{esize}$;

\text{CompareOp} \ \text{comparison};
case $\text{op:U}$ of
  when '00' $\text{comparison} = \text{CompareOp GT}$;
  when '01' $\text{comparison} = \text{CompareOp GE}$;
  when '10' $\text{comparison} = \text{CompareOp EQ}$;
  when '11' $\text{comparison} = \text{CompareOp LE}$;

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.

$<Vd>$ For the vector 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 vector 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 SIMD&FP source register, encoded in the "Rn" field.
Operation

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;
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)

```
<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 0</td>
</tr>
</tbody>
</table>
```

Scalar half precision

```
FCMGT <Hd>, <Hn>, <Hm>
```

if !HaveFP16Ext() then UNDEFINED;

```java
integer d = UInt(Rd);
ingen = 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 30 29 28 27 26 25 24 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</td>
</tr>
</tbody>
</table>
```

FCMGT (register)
Scalar single-precision and double-precision

FCMGT <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;
    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)

Vector half precision

FCMGT <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
Vector single-precision and double-precision

FCMGT \(<V_d>.<T>\), \(<V_n>.<T>\), \(<V_m>.<T>\)

integer \(d = \) UInt(Rd);
integer \(n = \) UInt(Rn);
integer \(m = \) UInt(Rm);
if \(sz:Q \equiv \) '10' then UNDEFINED;
integer esize = 32 \(<\) UInt(sz);
integer datasize = if \(Q \equiv \) '1' then 128 else 64;
integer elements = datasize DIV esize;

\(\text{CompareOp} \ cmp;\)
boolean abs;

\(\text{case} \ E:U:ac \ of\)
when '000' \(\text{cmp} = \text{CompareOp}_{\text{EQ}};\) abs = FALSE;
when '010' \(\text{cmp} = \text{CompareOp}_{\text{GE}};\) abs = FALSE;
when '011' \(\text{cmp} = \text{CompareOp}_{\text{GE}};\) abs = TRUE;
when '110' \(\text{cmp} = \text{CompareOp}_{\text{GT}};\) abs = FALSE;
when '111' \(\text{cmp} = \text{CompareOp}_{\text{GT}};\) abs = TRUE;
otherwise UNDEFINED;

Assembler Symbols

\(\text{<Hd}>\) 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 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.
\(\text{<V}>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>(sz)</th>
<th>(\text{&lt;V}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\(\text{<d}>\) Is the number of the SIMD&FP destination register, in the "Rd" field.
\(\text{<n}>\) Is the number of the first SIMD&FP source register, encoded in the "Rn" field.
\(\text{<m}>\) Is the number of the second SIMD&FP source register, encoded in the "Rm" field.
\(\text{<V_d}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(\text{<T}>\) For the vector half precision variant: is an arrangement specifier, encoded in "Q":

<table>
<thead>
<tr>
<th>(Q)</th>
<th>(\text{&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 vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>(sz)</th>
<th>(Q)</th>
<th>(\text{&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>

\(\text{<V_n}>\) Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
\(\text{<V_m}>\) 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 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;
```
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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0 |
| U  | op |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

Rn Rd

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  | 1  | sz| 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0 |
| U  | op |     |     |     |     |     |     | sz|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
Scalar single-precision and double-precision

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)

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

if !HaveFP16Ext() then UNDEFINED;

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 single-precision and double-precision

FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

```plaintext
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 vector 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 vector 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 = \text{V}[n];
bits(datasize) result;
bits(esize) zero = FPZero('0');
bits(esize) element;
boolean test_passed;

for \( e = 0 \) to elements-1
  element = \text{Elem}\{operand, e, esize\};
  case comparison of
    when \text{CompareOp GT} \hspace{0.2cm} \text{test passed} = FPCompareGT(element, zero, FPCR);
    when \text{CompareOp GE} \hspace{0.2cm} \text{test passed} = FPCompareGE(element, zero, FPCR);
    when \text{CompareOp EQ} \hspace{0.2cm} \text{test passed} = FPCompareEQ(element, zero, FPCR);
    when \text{CompareOp LE} \hspace{0.2cm} \text{test passed} = FPCompareGE(zero, element, FPCR);
    when \text{CompareOp LT} \hspace{0.2cm} \text{test passed} = FPCompareGT(zero, element, FPCR);
  \text{Elem}\{result, e, esize\} = \text{if test passed then Ones()} else Zeros();
\text{V}[d] = \text{result};
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  | L  | M  | Rm | 0  | rot | 1  | H  | 0  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Rd |

(size == 01)

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

<size == 10>

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>

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":

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;
<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**

```cpp
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
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, esize];
    when '10'
      element1 = FPNeg(Elem[operand2, index*2, esize]);
      element2 = Elem[operand1, e*2, esize];
      element3 = FPNeg(Elem[operand2, index*2+1, esize]);
      element4 = Elem[operand1, e*2+1, 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, 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;
```

FCMLA (by element)
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 | 0 | size | 0 | Rm | 1 | 1 | 0 | rot | 1 | Rn | Rd

Vector

FCMLA <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”: 
Operation

\begin{verbatim}
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;

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, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize]);
            element4 = Elem[operand1, e*2, 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;
\end{verbatim}

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 0 | 1 1 1 1 1 0 0 0 | 0 1 1 0 1 1 0 |
| U               | op              | Rd              |

Scalar half precision

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;
end;

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 1 sz 1 0 0 0 0 | 0 1 1 0 1 1 0 |
| U               | op              | Rd              |
Scalar single-precision and double-precision

FCMLE $<V><d>$, $<V><n>$, #0.0

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;

integer esize = 32 $\ll \text{UInt}(sz)$;
integer datasize = esize;
integer elements = 1;

\texttt{CompareOp} comparison;
case op:U of
  when '00' comparison = \texttt{CompareOp_GT};
  when '01' comparison = \texttt{CompareOp_GE};
  when '10' comparison = \texttt{CompareOp_EQ};
  when '11' comparison = \texttt{CompareOp_LE};

Vector half precision
\texttt{(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>U</td>
</tr>
</tbody>
</table>

Vector half precision

FCMLE $<Vd>\langle T\rangle$, $<Vn>\langle T\rangle$, #0.0

if !\texttt{HaveFP16Ext}() then UNDEFINED;

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;

integer esize = 16;
integer datasize = if $Q == '1'$ then 128 else 64;
integer elements = datasize $\div$ esize;

\texttt{CompareOp} comparison;
case op:U of
  when '00' comparison = \texttt{CompareOp_GT};
  when '01' comparison = \texttt{CompareOp_GE};
  when '10' comparison = \texttt{CompareOp_EQ};
  when '11' comparison = \texttt{CompareOp_LE};

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>
<tr>
<td>U</td>
</tr>
</tbody>
</table>
Vector single-precision and double-precision

FCMLE \(<Vd>.,<T>.,<Vn>.,<T>.\), \#0.0

integer \(d = \text{UInt}(\text{Rd});\)
integer \(n = \text{UInt}(\text{Rn});\)

if \(sz:Q == '10'\) then UNDEFINED;
integer \(\text{esize} = 32 << \text{UInt}(\text{sz});\)
integer \(\text{datasize} = \text{if}\ Q == '1' \text{then}\ 128\ \text{else}\ 64;\)
integer \(\text{elements} = \text{datasize} \div \text{esize};\)

\(\text{CompareOp}\ \text{comparison};\)
case \(\text{op}:U\) of
    when '00' \(\text{comparison} = \text{CompareOp\ GT};\)
    when '01' \(\text{comparison} = \text{CompareOp\ GE};\)
    when '10' \(\text{comparison} = \text{CompareOp\ EQ};\)
    when '11' \(\text{comparison} = \text{CompareOp\ LE};\)

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.
\(<Vd>\) For the vector half precision variant: is an arrangement specifier, encoded in "Q":

\(\text{Q} \quad <T>\)

<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 vector 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

```
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;
```
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 1 0 1 1 1 1 0 0 0 1 1 1 0 1 0</td>
</tr>
</tbody>
</table>

Scalar half precision

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 1</td>
</tr>
</tbody>
</table>

Scalar single-precision and double-precision

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 0 1 1 1 0 1 1 1 1 1 0 0 0 1 1 1 0 1 0</td>
</tr>
</tbody>
</table>
Vector half precision

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | sz | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | Rn | Rd |

Vector single-precision and double-precision

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

<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 vector 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 vector 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) 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);
    EndCase
    Elem[result, e, esize] = if test_passed then Ones() else Zeros();
```

`V[d] = result;`
FCMP

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 PSTATE.\{N, Z, C, V\} flags. It raises an Invalid Operation exception only if either operand is a signaling 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 0 1 1 1 1 0 ftype 1 0 0 1 0 0 0 Rn 0 x 0 0 0 opc

Half-precision (ftype == 11 && opc == 00) (Armv8.2)

FCMP <Hn>, <Hm>

Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 01) (Armv8.2)

FCMP <Hn>, #0.0

Single-precision (ftype == 00 && opc == 00)

FCMP <Sn>, <Sm>

Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 01)

FCMP <Sn>, #0.0

Double-precision (ftype == 01 && opc == 00)

FCMP <Dn>, <Dm>

Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 01)

FCMP <Dn>, #0.0

integer n = UInt(Rn);
integer m = UInt(Rm); // ignored when opc<0> == '1'

datasize;

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;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');


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.

NaNs
The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

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> = FPCmpare(operand1, operand2, signal_all_nans, FPCR);
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 PSTATE.\{N, Z, C, V\} flags.

If either operand is any type of NaN, or if either operand is a signaling NaN, the instruction raises an Invalid Operation exception.

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 1 1 1 1 0 ftype 1   |                           Rm |                           0 0 1 0 0 0 |                           Rn |   1 x | 0 0 0 0 |
| opc                          |-------------------------------|-------------------------------|-------------------------------|

Half-precision (ftype == 11 & opc == 10)

(Armv8.2)

FCMPE <Hn>, <Hm>

Half-precision, zero (ftype == 11 & Rm == (00000) & opc == 11)

(Armv8.2)

FCMPE <Hn>, #0.0

Single-precision (ftype == 00 & opc == 10)

FCMPE <Sn>, <Sm>

Single-precision, zero (ftype == 00 & Rm == (00000) & opc == 11)

FCMPE <Sn>, #0.0

Double-precision (ftype == 01 & opc == 10)

FCMPE <Dn>, <Dm>

Double-precision, zero (ftype == 01 & Rm == (00000) & opc == 11)

FCMPE <Dn>, #0.0

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
datasize = 16;
    else
      UNDEFINED;
  else
    UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
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.

\(<Dm>\) 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.

\(<Hm>\) 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.

\(<Sm>\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

NaNs
The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands are NaNs, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. This case results in the FPSCR flags being set to N=0, Z=0, C=1, and V=1.

FCMPE raises an Invalid Operation exception if either operand is any type of NaN, and is suitable for testing for <, <=, >, >=, and other predicates that raise an exception when the operands are unordered.

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);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 CPACR_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)

FCSEL \(<Hd>, <Hn>, <Hm>, <cond>\)

### Single-precision (ftype == 00)

FCSEL \(<Sd>, <Sn>, <Sm>, <cond>\)

### Double-precision (ftype == 01)

FCSEL \(<Dd>, <Dn>, <Dm>, <cond>\)

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);
integer \(m = \text{UInt}(Rm)\);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if \(\text{HaveFP16Ext}()\) then
      datasize = 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.
<\(cond>\) Is one of the standard conditions, encoded in the "cond" field in the standard way.
Operation

`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 FPCR 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.

<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 1 0</td>
</tr>
</tbody>
</table>

Half-precision to single-precision (ftype == 11 && opc == 00)

FCVT <Sd>, <Hn>

Half-precision to double-precision (ftype == 11 && opc == 01)

FCVT <Dd>, <Hn>

Single-precision to half-precision (ftype == 00 && opc == 11)

FCVT <Hd>, <Sn>

Single-precision to double-precision (ftype == 00 && opc == 01)

FCVT <Dd>, <Sn>

Double-precision to half-precision (ftype == 01 && opc == 11)

FCVT <Hd>, <Dn>

Double-precision to single-precision (ftype == 01 && opc == 00)

FCVT <Sd>, <Dn>

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;

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.
<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
Operation

```c
CheckFPAdvSIMDEnabled64();

bits(dstsize) result;
bits(srcsize) operand = V[n];

result = FPConvert(operand, FPCR);
V[d] = result;
```
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  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  |          | Rn | Rd |

Scalar half precision

FCVTAS <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  | 0  | 0  | 1  | 0  |          | Rn | Rd |

Scalar single-precision and double-precision

FCVTAS <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)

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

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 vector 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 vector 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**

\begin{verbatim}
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] = FPToFixed(element, 0, unsigned, FPCR, rounding);
V[d] = result;
\end{verbatim}
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.

![Instruction Format](image)

**Half-precision to 32-bit (sf == 0 && ftype == 11) (Armv8.2)**

```assembly
FCVTAS <Wd>, <Hn>
```

**Half-precision to 64-bit (sf == 1 && ftype == 11) (Armv8.2)**

```assembly
FCVTAS <Xd>, <Hn>
```

**Single-precision to 32-bit (sf == 0 && ftype == 00)**

```assembly
FCVTAS <Wd>, <Sn>
```

**Single-precision to 64-bit (sf == 1 && ftype == 00)**

```assembly
FCVTAS <Xd>, <Sn>
```

**Double-precision to 32-bit (sf == 0 && ftype == 01)**

```assembly
FCVTAS <Wd>, <Dn>
```

**Double-precision to 64-bit (sf == 1 && ftype == 01)**

```assembly
FCVTAS <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

CheckFPAdvSIMDEnabled64();
bits(fltsize) fltval;
bits(intsize) intval;
fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, FPRounding_TIEAWAY);
X[d] = intval;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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)*

![Encoded Format](image)

### Scalar half precision

**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

![Encoded Format](image)

### Scalar single-precision and double-precision

**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)

FCVTAU <Vd>.<T>, <Vn>.<T>

if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
in teger n = UInt(Rn);

integer esize = 16;
in teger datasize = if Q == '1' then 128 else 64;
in teger elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
b oolean unsigned = (U == '1');

Vector single-precision and double-precision

FCVTAU <Vd>.<T>, <Vn>.<T>

integer d = UInt(Rd);
in teger n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
in teger esize = 32 << UInt(sz);
in teger datasize = if Q == '1' then 128 else 64;
in teger elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
b oolean 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 vector 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 vector 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>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

\[
\text{CheckFPAdvSIMDEnabled64}(); \\
\text{bits(datasize) operand} = V[n]; \\
\text{bits(datasize) result}; \\
\text{bits(esize) element}; \\
\text{for } e = 0 \text{ to elements}-1 \\
\quad \text{element} = \text{Elem}[\text{operand}, e, \text{esize}]; \\
\quad \text{Elem}[\text{result, e, esize}] = \text{FPToFixed}(\text{element, 0, unsigned, FPCR, rounding}); \\
\text{V[d]} = \text{result};
\]
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.

<table>
<thead>
<tr>
<th>sf</th>
<th>0 0 1 1 1 1 0</th>
<th>ftype</th>
<th>1 0 0 1 0 1 0 0 0 0 0 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>rmode</td>
<td>opcode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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**

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, FPRounding_TIEAWAY);
X[d] = intval;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 |  0 | sz |  1 |  0 |  0 |  0 |  0 |  1 |  0 |  1 |  1 |  1 |  1 |  0 | Rn | Rd |

**Vector single-precision and double-precision**

FCVTL(2) <Vd>.<Ta>, <Vn>.<Tb>

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**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2</td>
<td></td>
</tr>
<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 “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>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" 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>25</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>45</td>
</tr>
</tbody>
</table>

**Operation**

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;
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 traps.

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

 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  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | Rn | Rd |
| U  | o2 | o1 |

Scalar half precision

 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | Rn | Rd |
| U  | o2 | o1 |

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 <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

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 vector 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 vector 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) element;

for e = 0 to elements-1
  element = Elem[operand, e, esize];
  Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

\[
\begin{array}{cccccccccccccccc}
\hline
sf & 0 & 0 & 1 & 1 & 1 & 1 & 0 & ftype & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & Rn & Rd \\
\end{array}
\]
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

```
CheckFPAdvSIMDEnabled64();

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 \texttt{FPCR}, the exception results in either a flag being set in \texttt{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \texttt{CPACR\_EL1}, \texttt{CPTR\_EL2}, and \texttt{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: \texttt{Scalar half precision}, \texttt{Scalar single-precision and double-precision}, \texttt{Vector half precision} and \texttt{Vector single-precision and double-precision}.

### Scalar half precision

\texttt{FCVTMU} <Hd>, <Hn>

\begin{verbatim}
if \texttt{!HaveFP16Ext()} then UNDEFINED;
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

\texttt{FPRounding} rounding = \texttt{FPDecodeRounding}(o1:o2);
boolean unsigned = (U == '1');
\end{verbatim}

### Scalar single-precision and double-precision

\texttt{FCVTMU} <V><d>, <V><n>

\begin{verbatim}
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer esize = 32 \texttt{UInt}(sz);
integer datasize = esize;
integer elements = 1;

\texttt{FPRounding} rounding = \texttt{FPDecodeRounding}(o1:o2);
boolean unsigned = (U == '1');
\end{verbatim}
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  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | Rn | Rd |

Vector half precision

FCVTMU <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  | Q  | 1  | 0  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | Rn | Rd |

Vector single-precision and double-precision

FCVTMU <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 vector 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 vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

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

<\text{n}> 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] = FPToFixed(element, 0, unsigned, FPCR, rounding);
```

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

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
31  30  29  28  27  26  25  24  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 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | Rn | Rd
  rmode  opcode
```
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 = 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();

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.

---

### Assembler Symbols

- **<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>4H</th>
<th>8H</th>
<th>2S</th>
<th>4S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></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 0 1 1 1 1 0 0 1 1 1 0 0 1 1 0 1 0 1 0 | Rn | Rd
U o2 o1
```

Scalar half precision

```
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 0 1 1 1 1 0 0 | sz | 1 0 0 0 0 1 1 0 1 0 1 0 | Rn | Rd
U o2 o1
```

Scalar single-precision and double-precision

```
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  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  |

Vector half precision

FCVTNS <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  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  |

Vector single-precision and double-precision

FCVTNS <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 vector 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 vector 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>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**

```plaintext
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] = FPToFixed(element, 0, unsigned, FPCR, rounding);
V[d] = result;
```

---

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | 1 | 0 | ftype | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
```

| rmode | opcode | Rd | Rn |
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 = 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

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = 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)

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |

Scalar half precision

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

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |

Scalar single-precision and double-precision

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)

```
  31  30  29  28  27  26  25  24  23  22  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  | 0  | 0  | 1  | 0  | 1  | 0  | Rn   
U   o2 o1   1

Vector half precision
```

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 = FPD\text{DecodeRounding}(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  | 0  | 1  | 1  | 1  | 0  | 0  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | Rn   
U   o2 o1   1

Vector single-precision and double-precision
```

FCVTNU <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 = FPD\text{DecodeRounding}(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>` is the vector 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 vector 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>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**

\begin{verbatim}
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] = FPToFixed(element, 0, unsigned, FPCR, rounding);
V[d] = result;
\end{verbatim}
**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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  |
| rmode | opcode |

*sf*: Sign fraction
*ftype*: Floating-point type
*Rn* and *Rd*: General-purpose registers
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();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 \texttt{FPCR}, the exception results in either a flag being set in \texttt{FPSR}, or a synchronous exception being generated. For more information, see \textit{Floating-point exception traps}.

Depending on the settings in the \texttt{CPACR EL1}, \texttt{CPTR EL2}, and \texttt{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: \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})

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 0  | 1  | 0  | Rn |

\texttt{U} \quad \texttt{o2} \quad \texttt{o1} \quad \texttt{Rd}

**Scalar half precision**

FCVTPS <\texttt{Hd}>\, <\texttt{Hn}>

\begin{verbatim}
if !\texttt{HaveFP16Ext}() then UNDEFINED;

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

\texttt{FPRounding} rounding = \texttt{FPDecodeRounding}(o1:o2);
boolean unsigned = (U == '1');
\end{verbatim}

### 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  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | Rn |

\texttt{U} \quad \texttt{o2} \quad \texttt{o1} \quad \texttt{Rd}

**Scalar single-precision and double-precision**

FCVTPS <\texttt{V}<d>, <V><n>>

\begin{verbatim}
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer esize = 32 \ll \texttt{UInt}(sz);
integer datasize = esize;
integer elements = 1;

\texttt{FPRounding} rounding = \texttt{FPDecodeRounding}(o1:o2);
boolean unsigned = (U == '1');
\end{verbatim}
**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  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |

**Vector half precision**

`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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | sz | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | Rn | Rd |

**Vector single-precision and double-precision**

`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 vector 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 vector 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
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] = FPToFixed(element, 0, unsigned, FPCR, rounding);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | 0 | ftype | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
```

The **opcode** is generated using the function code **0x28**.
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 = 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();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToF1xed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;
FCVTPU (vector)

Floating-point Convert toUnsigned 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 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 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
|   | U | o2 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Scalar half precision

FCVTPU <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 | 1 | 1 | 0 | 1 | sz | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
|   | U | o2 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Scalar single-precision and double-precision

FCVTPU <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)

| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |

Vector half precision

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

| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | Rn | Rd |
| U | o2 | o1 |

Vector single-precision and double-precision

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>&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 vector 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 vector 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 = \text{V}[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, FPCR, rounding);

V[d] = result;
```

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 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)

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 = 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();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = 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

<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>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>0</td>
<td>1</td>
<td>1</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>
<td></td>
</tr>
</tbody>
</table>

Scalar

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

<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>0</td>
<td>1</td>
<td>1</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>
<td></td>
</tr>
</tbody>
</table>

Vector

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

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".
<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>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>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>S</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];
bits(datasize) result;
for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR, FPRounding_ODD);
Vpart[d, part] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 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 1 0 | != 0000 | immh | 1 1 1 1 1 1 | Rn | Rd
  U

Scalar

FCVTZS <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 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
 0 | Q | 0 0 1 1 1 1 0 | != 0000 | immh | 1 1 1 1 1 1 | Rn | Rd
  U

Vector

FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>

integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SFE(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

<Y> Is a width specifier, encoded in “immh”:
<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>001 x</td>
<td>x</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</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>

<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>0000</td>
<td>x</td>
</tr>
<tr>
<td>001 x</td>
<td>x</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
</tr>
<tr>
<td>01xx</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>

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>x</td>
</tr>
<tr>
<td>001 x</td>
<td>x</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>

**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];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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)

<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 0 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 1 0 0 1 1 0 1 1 0</td>
</tr>
<tr>
<td>U o2             Rn          Rd</td>
</tr>
</tbody>
</table>

**Scalar half precision**

FCVTZS <Hd>, <Hn>

if ![HaveFP16Ext()]() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

d = 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 0 1 1 1 0 0 1 1 0 0 0 0 1 1 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 0 1 1 0 0 0 0 1 1 1 0 1 1 1 0 1 1 0 0 1 1 0 1 1 0</td>
</tr>
</tbody>
</table>

**Scalar single-precision and double-precision**

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)

Vector half precision

FCVTZS \langle Vd\rangle,\langle T\rangle, \langle Vn\rangle,\langle T\rangle

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

FCVTZS \langle Vd\rangle,\langle T\rangle, \langle Vn\rangle,\langle T\rangle

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

\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 vector half precision variant: 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>4H</td>
</tr>
<tr>
<td>1</td>
<td>8H</td>
</tr>
</tbody>
</table>

FCVTZS (vector, integer)
For the vector 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>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 = \textit{V}[n];
bits(datasize) result;
bits(esize) element;
for \( e = 0 \) to elements-1
    element = \textit{Elem}[operand, e, esize];
    \textit{Elem}[result, e, esize] = \textit{FPToFixed}(element, 0, unsigned, FPCR, rounding);
\textit{V}[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmode</td>
<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>
</tr>
</tbody>
</table>

**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>
```

```plaintext
d = UInt(Rd);
n = UInt(Rn);

int intsize = if sf == '1' then 64 else 32;
int 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;
int 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();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, fracbits, FALSE, FPCR, FPRounding_ZERO);
X[d] = intval;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | 0 | ftype | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|    |   |   |   |   |   |   |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|    |   |   |   |   |   |   |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|    |   |   |   |   |   |   |       |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

opcode  
rmode  
Rn  
Rd  

Page 669
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();`

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, 0, FALSE, FPCR, rounding);
X[d] = intval;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

<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>

Scalar

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 datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;

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>

Vector

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 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

<V> Is a width specifier, encoded in “immh”:
immh | <V>
---|---
000x | RESERVED
001x | H
01xx | S
1xxx | 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.
<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_modifiedImmediate</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>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_modifiedImmediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01lx</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) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, FPCR, rounding);

V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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

**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 | o2 | o1 |

#### Scalar half precision

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

| 31 30 29 28 27 26 25 24 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 | o2 | o1 |

#### Scalar single-precision and double-precision

FCVTZU `<VD><D>, <VN><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)

Vector half precision

FCVTZU <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

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 vector 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 vector 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**

\begin{verbatim}
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] = FPToFixed(element, 0, unsigned, FPCR, rounding);
V[d] = result;
\end{verbatim}

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.

```markdown
|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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|    |    | 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>

```markdown
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();

bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n];
intval = FPToFixed(fltval, fracbits, TRUE, FPCR, FPRounding_ZERO);
X[d] = intval;
```

Copyright © 2010-2019 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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | Rn |    |    |    |    |    |    |    |    |    |    |
```

rmode   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 = 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();

bits(fltsize) fltval;
bites(intsize) intval;

fltval = V[n];
intval = FPtoFixed(fltval, 0, TRUE, FPCR, rounding);
X[d] = intval;
**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)**

```
0 0 1 1 1 0 | 0 1 0 | Rm
0 0 1 1 1 1 | 1 1 | Rn
1 1 | Rd
```

**Half-precision**

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

```
0 0 1 1 1 0 | 0 | sz 1 | Rm
1 1 1 1 1 1 | 1 | Rn
1 1 | Rd
```

**Single-precision and double-precision**

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>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>

**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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

```
0 0 0 1 1 1 0 1 ftype 1 Rm 0 0 0 1 1 0 Rn 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 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
```

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

result = FPDiv(operand1, operand2, FPCR);
V[d] = result;
FJCVTZS

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 accommodated 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)

Integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveFJCVTZSExt() then UNDEFINED;

Assembler Symbols

<Wnd> 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();

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;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

```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  |   | ftype | 0  |    | Rm  | 0  |    | Ra  |    |    | Rn  |    |    | Rd  |    |    |    |    |    |    |    |    |    |
```

### Half-precision (ftype == 11)

(ARMv8.2)

```plaintext
FMADD <Hd>, <Hn>, <Hm>, <Ha>
```

### Single-precision (ftype == 00)

```plaintext
FMADD <Sd>, <Sn>, <Sm>, <Sa>
```

### Double-precision (ftype == 01)

```plaintext
FMADD <Dd>, <Dn>, <Dm>, <Da>
```

```plaintext
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
\text{case ftype of}
\text{when '00' datasize = 32;}
\text{when '01' datasize = 64;}
\text{when '10' UNDEFINED;}
\text{when '11'}
\text{if HaveFP16Ext() then}
\text{datasize = 16;}
\text{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.

<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(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
result = FPMulAdd(operanda, operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 1  | 0  | 1  | Rd |
| U  | o1 |

**Half-precision**

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | Q  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | sz | 1  | Rm | 1  | 1  | 1  | 0  | 1  | Rd |
| U  | o1 |

**Single-precision and double-precision**

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”: 

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
CheckFPAvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(2*esize) element1;
bits(2*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 v31.04, AdvSIMD v29.02, pseudocode v2019-09 rc2_1, sve v2019-09 rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 1 0 ftype 1 Rm 0 1 0 0 1 0 Rn Rd

op

Half-precision (ftype == 11)
(Armv8.2)

FMAX <Hd>, <Hn>, <Hm>

Single-precision (ftype == 00)

FMAX <Sd>, <Sn>, <Sm>

Double-precision (ftype == 01)

FMAX <Dd>, <Dn>, <Dm>

integer d = UInt(Rd);
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;

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

result = FMax(operand1, operand2, FPCR);
V[d] = result;
```
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  | Q | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
| U  | a |

**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 | 0 | 0 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| U  | 01 |

**Assembler Symbols**

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

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');
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();
b bits(datasize) operand1 = V[n];
b bits(datasize) operand2 = V[m];
b bits(datasize) result;
b bits(2*datasize) concat = operand2:operand1;
b bits(esize) element1;
b 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | 0  | 1  | 1  | 1  | 1  | 0  | ftype | 1  | Rm | 0  | 1  | 1  | 0  | 1  | Rd | 0  |

op

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 datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 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(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMaxNum(operand1, operand2, FPCR);
V[d] = result;
\end{verbatim}

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \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, 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: \textcolor{red}{Half-precision} and \textcolor{blue}{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  0 | 0  1  1  0  0  | Rn  | Rd
  0  1  sz
\end{verbatim}

\textbf{Half-precision}

FMAXNMP $<V><d>, <Vn>.<T>

\begin{verbatim}
if !HaveFP16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
integer datasize = 32;
\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  1  1 | 1  1  1  0  0  0 | 0  1  1  0  0  | Rn  | Rd
  0  1
\end{verbatim}

\textbf{Single-precision and double-precision}

FMAXNMP $<V><d>, <Vn>.<T>

\begin{verbatim}
integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 32;
integer datasize = 64;
\end{verbatim}

\textbf{Assembler Symbols}

\begin{verbatim}<V>
\end{verbatim}

For the half-precision variant: is the destination width specifier, encoded in “sz”:

\begin{verbatim}
<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>
\end{verbatim}

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

\begin{verbatim}
<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>
\end{verbatim}

\begin{verbatim}<d>
\end{verbatim}

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_FMAXNUM, operand, esize);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 0  | 0  | 0  | 1  | Rn | Rd |

**U** | \text{a}

### 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  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 0  | 0  | 1  | Rn | Rd |

**U** | \text{0l}

### Single-precision and double-precision

```
FMAXNMP <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;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

```
FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);
integer m = \text{UInt}(Rm);
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;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```
Assembler Symbols

\(<\text{Vd}\>\) Is the name of the SIMD\&FP destination register, encoded in the "Rd" field.

\(<\text{T}\>\) 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>

\(<\text{Vn}\>\) 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2.1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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)

```
0 1 1 1 0 | 0 0 1 1 0 0 0 0 1 1 0 0 1 0 | Rn   | Rd
```

**Half-precision**

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**

```
0 1 1 1 0 | 0 1 1 0 0 0 0 1 1 0 0 1 0 | Rn   | Rd
```

**Single-precision and double-precision**

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>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_FMAXNUM, operand, esize);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2.1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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)

```
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**

```
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.
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_FMAX, operand, esize);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 1  | 1  | 0  | 1  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Rd |

#### Half-precision

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  | 0  | 0  | 1  | 0  | sz | Rm | 1  | 1  | 1  | 0  | 1  | Rn |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | Rd |

#### Single-precision and double-precision

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":
For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2S</td>
<td>2D</td>
</tr>
<tr>
<td>T</td>
<td>14S</td>
<td>12D</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>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2S</td>
<td>2D</td>
</tr>
<tr>
<td>T</td>
<td>14S</td>
<td>12D</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>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>2S</td>
<td>2D</td>
</tr>
<tr>
<td>T</td>
<td>14S</td>
<td>12D</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**

```pseudo
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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)

```
0 Q 0 0 1 1 1 0 0 1 1 0 0 0 1 1 1 1 1 0
```

```
Rn
Rd
```

Half-precision

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

```
0 Q 1 0 1 1 1 0 0 sz 1 1 0 0 0 0 1 1 1 1 1 0
```

```
Rn
Rd
```

Single-precision and double-precision

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”:

```
sz <V>

<table>
<thead>
<tr>
<th>0</th>
<th>S</th>
</tr>
</thead>
<tbody>
<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.
For the half-precision variant: is an arrangement specifier, encoded in "Q":

<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></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 "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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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)**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    | Rm|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Half-precision

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 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  |
|    |    | Rm|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Single-precision and double-precision

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: 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();
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;
```
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 1  | 1  | 0  | Rn | Rd |

op

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>

integer d = UInt(Rd);
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;

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

result = 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 | 0 | 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 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 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.
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] = FPMinNum(element1, element2, FPCR);
  else
    Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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, 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  | 1  | 1  | 0  |
  |   |   | ftype |   |   |   | op |
31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
```

**Half-precision (ftype == 11)** (Armv8.2)

\[ FMINNM \ <Hd>, <Hn>, <Hm> \]

**Single-precision (ftype == 00)**

\[ FMINNM \ <Sd>, <Sn>, <Sm> \]

**Double-precision (ftype == 01)**

\[ FMINNM \ <Dd>, <Dn>, <Dm> \]

```
integer d = UInt(Rd);
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
data size = 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(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMinNum(operand1, operand2, FPCR);
V[d] = result;
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  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |
| o1  | sz |

Half-precision

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  | 0  | 1  | sz | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | Rn | Rd |
| o1  |    |

Single-precision and double-precision

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.
<\text{Vn}> \quad \text{Is the name of the SIMD\&FP source register, encoded in the "Rn" field.}
<\text{T}> \quad \text{For the half-precision variant: is the source arrangement specifier, encoded in "sz":}

\begin{tabular}{|c|c|}
\hline
\text{sz} & \text{<\text{T}>} \\
\hline
0 & 2H \\
1 & RESERVED \\
\hline
\end{tabular}

\text{For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":}

\begin{tabular}{|c|c|}
\hline
\text{sz} & \text{<\text{T}>} \\
\hline
0 & 2S \\
1 & 2D \\
\hline
\end{tabular}

\textbf{Operation}

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \text{V}[n];
\text{V}[d] = \text{Reduce(\text{ReduceOp_FMINNUM}, operand, esize)};
\end{verbatim}

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 | 0 | Q | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | Rm | 0 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
| U | a |

**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 | Q | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | sz | 1 | Rm | 1 | 1 | 0 | 0 | 0 | 1 | Rn | Rd |
| U | 0 | l |

**Single-precision and double-precision**

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');
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] = FPMINNum(element1, element2, FPCR);
    else
        Elem[result, e, esize] = FPMAXNum(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point Minimum Number 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.

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 \textit{FMIN (scalar)}. 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 2 classes: \textit{Half-precision} and \textit{Single-precision and double-precision}.

**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
| Q | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd |
01
\end{verbatim}

**Half-precision**

\begin{verbatim}
FMINNMV <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;
\end{verbatim}

**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
| Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | sz | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | Rn | Rd |
01
\end{verbatim}

**Single-precision and double-precision**

\begin{verbatim}
FMINNMV <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;
\end{verbatim}

**Assembler Symbols**

\begin{verbatim}<V>\end{verbatim} For the half-precision variant: is the destination width specifier, H.
\begin{verbatim}<d>\end{verbatim} Is the number of the SIMD&FP destination register, encoded in the “Rd” field.

\begin{verbatim}<V>\end{verbatim} For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:
\begin{verbatim}
sz <V>
0 S
1 RESERVED
\end{verbatim}
<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_FMINNUM, operand, esize);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 \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}

**Half-precision**

\textit{(Armv8.2)}

\[
\begin{array}{cccccccccccccccccccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & Rn & Rd \\
\end{array}
\]

01 sz

**Half-precision**

\texttt{FMINP <V><d>, <Vn>.<T>}

\texttt{if !HaveFP16Ext() then UNDEFINED;}

\texttt{integer d = UInt(Rd);}
\texttt{integer n = UInt(Rn);}

\texttt{integer esize = 16;}
\texttt{integer datasize = 32;}

**Single-precision and double-precision**

\[
\begin{array}{cccccccccccccccccccccccc}
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & Rn & Rd \\
\end{array}
\]

01

**Single-precision and double-precision**

\texttt{FMINP <V><d>, <Vn>.<T>}

\texttt{integer d = UInt(Rd);}
\texttt{integer n = UInt(Rn);}

\texttt{integer esize = 32;}
\texttt{integer datasize = 64;}

**Assembler Symbols**

\(<V>\quad\text{For the half-precision variant: is the destination width specifier, encoded in "sz":}\)

\[
\begin{array}{c c}
sz & <V> \\
0 & H \\
1 & RESERVED \\
\end{array}
\]

\text{For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:}

\[
\begin{array}{c c}
sz & <V> \\
0 & S \\
1 & D \\
\end{array}
\]

\(<d>\quad\text{Is the number of the SIMD&FP destination register, encoded in the "Rd" field.}\)

\(<Vn>\quad\text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}\)
For the half-precision variant: is the source arrangement specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>sz &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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 &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
V[d] = Reduce(ReduceOp_FMIN, operand, esize);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 | Rm | 0 | 0 | 1 | 1 | 0 | 1 | Rn | Rd |

**Half-precision**

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 = (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 | 0 | 1 | 1 | 0 | sz | 1 | Rm | 1 | 1 | 1 | 0 | 1 | Rn | Rd |

**Single-precision and double-precision**

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 = (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":

FMINP (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>

\(<V_n>\) 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) 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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)

<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>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>1</td>
<td>1</td>
<td>1</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Half-precision

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

<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>0</td>
<td>1</td>
<td>sz</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>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Single-precision and double-precision

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”:

<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.
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_FMIN, operand, esize);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | 0 1 0 1 1 1 1 1 0 0 | L | M | Rm | 0 | 0 | 0 | 1 | H | 0 | Rn | Rd |
```

Scalar, half-precision

FMLA <Hd>, <Hn>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

```
integer idxsiz = 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  | 0 1 0 1 1 1 1 1 1 | sz | L | M | Rm | 0 | 0 | 0 | 1 | H | 0 | Rn | Rd |
```
Scalar, single-precision and double-precision

FMLA $<V><d>$, $<V><n>$, $<Vm>.<T>[_{<index>}]$

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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | L  | M  | Rm | 0  | 0  | 0  | 1  | H  | 0  | Rn | Rd |
| 02 |

Vector, half-precision

FMLA $<Vd>.<T>$, $<Vn>.<T>$, $<Vm>.H[_{<index>}]$

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 = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '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  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | sz | L  | M  | Rm | 0  | 0  | 0  | 1  | H  | 0  | Rn | Rd |
| 02 |
Vector, single-precision and double-precision

FMLA <Vd>.<T>, <Vn>.<T>, <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);
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 = (o2 == '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":

<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.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the vector, 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 vector, 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".

FMLA (by element) Page 731
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
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;
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)

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | 1 | 0 | Rm | 0 | 0 | 0 | 0 | 1 | 1 | Rn | Rd |

**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

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | 0 | 1 | 0 | sz | 1 | Rn | 1 | 1 | 0 | 0 | 1 | 1 | Rd |

**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";
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) 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 AA64ISARO_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
Q | L | M | Rm | O | 0 | 0 | 0 | H | 0 | Rn | Rd
```

### FMLAL

FMLAL <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;

### 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
Q | L | M | Rm | 1 | 0 | 0 | 0 | H | 0 | Rn | Rd
```

### FMLAL, FMLAL2 (by element)

Page 735
FMLAL2 <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 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.
<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

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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)

<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>S</td>
</tr>
</tbody>
</table>

FMLAL

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)

<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>S</td>
</tr>
</tbody>
</table>

FMLAL2

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":

<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.

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)

```
31 30 29 28 27 26 25 24 23 22 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 | 1 | 0 | 1 | H | 0 | Rn | Rd |
   02
```

**Scalar, half-precision**

FMLS <Hd>, <Hn>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsiz = 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 | 1 | 0 | 1 | H | 0 | Rn | Rd |
   02
```
Scalar, single-precision and double-precision

FMLS \(<V><d>, <V><n>, <Vm><Ts>[<index>]\)

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)

31 30 29 28 27 26 25 24 23 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 | L | M | Rm | 0 | 1 | 0 | 1 | H | 0 | Rn | Rd | 02

Vector, half-precision

FMLS \(<Vd><T>, <Vn><T>, <Vm>.H[<index>]\)

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 = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '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 | Q | 0 | 0 | 1 | 1 | 1 | 1 | sz | L | M | Rm | 0 | 1 | 0 | 1 | H | 0 | Rn | Rd | 02

FMLS (by element)
Vector, single-precision and double-precision

FMLS \(<Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]\)

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;

d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV esize;
boolean sub_op = (o2 == '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":

<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.
\(<Vd>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<T>\) For the vector, 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 vector, 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":

Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) operand3 = V[d];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q  | Rm | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | Rd |

### Half-precision

FMLS <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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | Rm | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | Rd |

### Single-precision and double-precision

FMLS <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”:
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) 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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             | Q             | 0             | 0             | 1             | 1             | 1             | 1             | 0             | L             | M             | Rm            | 0             | 1             | 0             | 0             | H             | 0             | Rn            | Rd            |

**FMLSL**

<Fd>.<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             | Q             | 1             | 0             | 1             | 1             | 1             | 1             | 1             | 1             | 0             | L             | M             | Rm            | 1             | 1             | 0             | 0             | H             | 0             | Rn            | Rd            |

---

FMLSL, FMLSL2 (by element)
FMLSL2

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.
<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

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;
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 | Rm | 1 1 1 0 1 1 | Rn | Rd
  S  sz
```

FMLSL

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 | Rm | 1 1 0 0 1 1 | Rn | Rd
  S  sz
```

FMLSL2

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.
<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.
<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

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;
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 CPACR_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  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | a  | b  | c  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | d  | e  | f  | g  | h  | Rd |

**Half-precision**

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);

### 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 | op | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | a  | b  | c  | 1  | 1  | 1  | 1  | 0  | 1  | d  | e  | f  | g  | h  | Rd |

cmode

**Single-precision (op == 0)**

FMOV <Vd>.<T>, #<imm>

**Double-precision (Q == 1 & & op == 1)**

FMOV <Vd>.2D, #<imm>

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;

bits(datasize) imm;

bits(64) imm64;

if cmode:op == '11111' then

  // FMOV Dn,#imm is in main FP instruction set
  if Q == '0' then UNDEFINED;

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.
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 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>2S</td>
</tr>
<tr>
<td>1</td>
<td>4S</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**

```
CheckFPAdvSIMDEnabled64();
V[rd] = imm;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

<table>
<thead>
<tr>
<th>FMOV (register)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&amp;FP source register to the SIMD&amp;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.</td>
</tr>
</tbody>
</table>

Half-precision (ftype == 11)
(ARMv8.2)

FMOV <Hd>, <Hn>

Single-precision (ftype == 00)

FMOV <Sd>, <Sn>

Double-precision (ftype == 01)

FMOV <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

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;

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();

bits(datasize) operand = V[n];
V[d] = operand;

Copyright © 2010-2019 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.

```
<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>x</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>
<tr>
<td>rmode</td>
<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>
</tr>
</tbody>
</table>
```
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;
    end;
  end;

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;
end;

---

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

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

case op of
  when FPConvOp_CVT_FtoI
      fltval = V[n];
      intval = FPtoFixed(fltval, 0, unsigned, FPCR, rounding);
      X[d] = intval;
  when FPConvOp_CVT_ItoF
      intval = X[n];
      fltval = FixedToFP(intval, 0, unsigned, FPCR, rounding);
      V[d] = fltval;
  when FPConvOp_MOV_FtoI
      fltval = Vpart[n, part];
      intval = ZeroExtend(fltval, intsize);
      X[d] = intval;
  when FPConvOp_MOV_ItoF
      intval = X[n];
      fltval = intval<fltsize-1:0>;
      Vpart[d, part] = fltval;
  when FPConvOp_CVT_FtoI_JS
      bit Z;
      fltval = V[n];
      (intval, Z) = FPtoFixedJS(fltval, FPCR, TRUE);
      PSTATE.<N,Z,C,V> = '0':Z:'00';
      X[d] = intval;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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, CPTTR_EL2, and CPTTR_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)
  
  FMOV <Hd>, #<imm>

- **Single-precision (ftype == 00)**
  
  FMOV <Sd>, #<imm>

- **Double-precision (ftype == 01)**
  
  FMOV <Dd>, #<imm>

```plaintext
integer d = UInt(Rd);
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(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Floating-point Fused Multiply-Add (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.

### 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(datasize) result;
bits(datasize) operand = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operand1 = FPNeg(operand1);
result = FPMulAdd(operand, operand1, operand2, FPCR);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 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 | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |

Scalar, half-precision

FMUL <Hd>, <Hn>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsze = 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | sz | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn | Rd |
**Scalar, single-precision and double-precision**

\[
\text{FMUL } <V><d>, <V><n>, <Vm>.<Ts>[<index>]
\]

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 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  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn |  |   | Rd |   |

**Vector, half-precision**

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | Q  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | sz | L  | M  | Rm | 1  | 0  | 0  | 1  | H  | 0  | Rn |  |   | Rd |   |
# Vector, single-precision and double-precision

\[ \text{FMUL} \ <V_d>,<T>, <V_n>,<T>, <V_m>,<T_s>[<\text{index}>] \]

```plaintext
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":
  ```plaintext
<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.
- `<Vd>`: Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
- `<T>`: For the vector, half-precision variant: is an arrangement specifier, encoded in "Q":
  ```plaintext
<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 vector, single-precision and double-precision variant: is an arrangement specifier, encoded in "Q:sz":
  ```plaintext
<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":
  ```plaintext
<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":
```
### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxdsize) operand2 = V[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];

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;
```

---

FMUL (by element)
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  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Rm | 0  | 0  | 0  | 1  | 1  | Rn | Rd |

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  | 1  | 0  | 0  | sz | 1  | Rm | 1  | 1  | 0  | 1  | 1  | 1  | Rn | Rd |

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(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] = FPMul(element1, element2, FPCR);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 | Rd  | 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>

```plaintext
dm = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

dataSize;

case ftype of
  when '00' dataSize = 32;
  when '01' dataSize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      dataSize = 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

\textbf{CheckFPAdvSIMDEnabled64()};
\begin{verbatim}
  bits(datasize) result;
  bits(datasize) operand1 = V[n];
  bits(datasize) operand2 = V[m];

  result = FMul(operand1, operand2, FPCR);

  V[d] = result;
\end{verbatim}
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 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 1 1 1 1 1 1 1 1 0 0 | L | M | Rm | 1 0 0 1 | H | 0 | Rn | Rd |

Scalar, half-precision

FMULX <Hd>, <Hn>, <Vm>.H[index]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsze = 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 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 1 1 1 1 1 1 1 1 1 | sz | L | M | Rm | 1 0 0 1 | H | 0 | Rn | Rd |
Scalar, single-precision and double-precision

FMULX <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 mulx_op = (U == '1');

Vector, half-precision
(Armv8.2)

Vector, half-precision

FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
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

Vector, single-precision and double-precision

FMULX (by element)
Vector, single-precision and double-precision

FMULX <Vd>.<T>, <Vn>.<T>, <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);
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":

<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.
<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T> For the vector, 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 vector, 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":

FMULX (by element)
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsiz) operand2 = V[m];
bits(datasize) result;
bits(esez) element1;
bits(esez) element2 = Elem[operand2, index, esize];
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;
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)

<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 1 0</td>
</tr>
</tbody>
</table>

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 0 sz 1</td>
</tr>
</tbody>
</table>

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 0 V</td>
</tr>
</tbody>
</table>

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 0 1 1 1 0 0 1 0</td>
</tr>
</tbody>
</table>
Vector half precision

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

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

<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.

<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 vector 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 vector 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();`

```c
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] = \$FPMulX(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 CPACR_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>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>0</td>
<td>0</td>
<td>1</td>
<td>1</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>
</tr>
</tbody>
</table>

Rn Rd U

Half-precision

FNEG <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</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>0</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>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>
</tr>
</tbody>
</table>

Rn Rd U

Single-precision and double-precision

FNEG <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”:

<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”:
<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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09 rc2_1, sve v2019-09 rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

![Operand Encoding](image)

**Half-precision (ftype == 11)**
(Armv8.2)

\[
FNEG <Hd>, <Hn>
\]

**Single-precision (ftype == 00)**

\[
FNEG <Sd>, <Sn>
\]

**Double-precision (ftype == 01)**

\[
FNEG <Dd>, <Dn>
\]

integer d = UInt(Rd);
integer n = UInt(Rn);

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;

Assembler Symbols

\[
<Dd> \quad \text{Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
<Dn> \quad \text{Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
\]

\[
< Hd > \quad \text{Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
<Hn> \quad \text{Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
\]

\[
<Sd> \quad \text{Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.}
\]

\[
<Sn> \quad \text{Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.}
\]

Operation

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

bits(datasize) result;
bits(datasize) operand = V[n];
result = FPNeg(operand);
V[d] = result;
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.

```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 1 ftype 1 | Rm 0 | Ra | Rn | Rd
   01 | o0
```

**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
d = UInt(Rd);
a = UInt(Ra);
n = UInt(Rn);
m = UInt(Rm);

datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 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.
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**

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operanda = FPNeg(operanda);
operand1 = FPNeg(operand1);
result = FPMulAdd(operanda, operand1, operand2, FPCR); 

V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | ftype | 1  | Rm | 1  | Ra | 1  | Rn | 1  | Rd | 0  | 1  | 0  |

**Half-precision (ftype == 11)**

(Armv8.2)

`FNMSUB <Hd>, <Hn>, <Hm>, <Ha>`

**Single-precision (ftype == 00)**

`FNMSUB <Sd>, <Sn>, <Sm>, <Sa>`

**Double-precision (ftype == 01)**

`FNMSUB <Dd>, <Dn>, <Dm>, <Da>`

```plaintext
integer d = UInt(Rd);
integer a = UInt(Ra);
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;
```

**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(datasize) result;

bits(datasize) operanda = V[a];
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

operanda = FPNeg(operanda);
result = FPMulAdd(operanda, operand1, operand2, FPCR);

V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 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 datasize;
case ftype of
when '00' datasize = 32;
when '01' datasize = 64;
when '10' UNDEFINED;
when '11'
    if HaveFP16Ext() then
datasize = 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(datasize) result;
bits(datasize) operand1 = V[n];
bits(datasize) operand2 = V[m];

result = FPMul(operand1, operand2, FPCR);
result = FPNeg(result);
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)

<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>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>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>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar half precision

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

<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>1</td>
<td>sz</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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar single-precision and double-precision

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)

<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>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>1</td>
<td>1</td>
<td>0</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>Rn</td>
<td>Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector half precision

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: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 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 vector 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 vector 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] = 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)

```
31 30 29 28 27 26 25 24 23 22 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 1 1 1 1 | Rn  | Rd
```

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 0 0 sz 1 | Rm 1 1 1 1 1 1 | Rn  | Rd
```

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 = 16;
integer datasize = esize;
integer 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 0 0 1 1 1 0 0 1 0 | Rm 0 0 1 1 1 1 | Rn  | Rd
```

VRECPS
Vector half precision

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

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>&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 vector 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 vector 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(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] = FPRecipStepFused(element1, element2);
V[d] = 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 | 1 | 0 | 0 | 1 | 1 | 1 | 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 | 1 | 0 | 0 | 0 | 0 | 1 | 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>

- **<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.
**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];
  Elem[result, e, esize] = FPREqpX(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_EL2, 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

(Vector single-precision and double-precision)

FRINT32X <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.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" 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>

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;
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  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |

ftype     op

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 datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 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();

bits(datasize) result;
bits(datasize) operand = V[n];

result = 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 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 0 0 0 1 1 1 0 0 0 sz 1 0 0 0 0 1 1 1 1 0 1 0 | Rn | Rd |
|    U    |    op    |

Vector single-precision and double-precision

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 round = 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.
<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;
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)

<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 1 0 0 x 1 0 1 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>ftype op Rn Rd</td>
</tr>
</tbody>
</table>

Single-precision (ftype == 00)

FRINT32Z <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT32Z <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 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.
<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();

bits(datasize) result;
bits(datasize) operand = V[n];

result = 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 | sz | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| U | op | Rd |

Vector single-precision and double-precision

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.
<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<table>
<thead>
<tr>
<th>sz Q &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 25</td>
</tr>
<tr>
<td>0 1 45</td>
</tr>
<tr>
<td>1 0 RESERVED</td>
</tr>
<tr>
<td>1 1 20</td>
</tr>
</tbody>
</table>

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  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | Rn | Rd |

ftype  op

Single-precision (ftype == 00)

FRINT64X <Sd>, <Sn>

Double-precision (ftype == 01)

FRINT64X <Dd>, <Dn>

if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;
case ftype of
when '00' datasize = 32;
when '01' datasize = 64;
when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR);

Assembler Symbols

<Dr> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dr> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dr> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];
result = 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)

<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>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>0</td>
<td>Rn</td>
<td>op</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>

### Vector single-precision and double-precision

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;
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 \(\text{corresponding}\) 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 \(\text{FPCR}\), 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 \(\text{CPACR\_EL1, CPTR\_EL2, and CPTR\_EL3}\) 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  |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| \text{ftype} \text{op} | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | x  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | Rn | Rd |

Single-precision (\text{ftype} == 00)

FRINT64Z \(<\text{Sd}>, <\text{Sn}>\)

Double-precision (\text{ftype} == 01)

FRINT64Z \(<\text{Dd}>, <\text{Dn}>\)

\[
\text{if} \quad \text{!HaveFrintExt()} \quad \text{then UNDEFINED;}
\]

\[
\text{integer} \quad d = \text{UInt}(\text{Rd});
\]

\[
\text{integer} \quad n = \text{UInt}(\text{Rn});
\]

\[
\text{integer} \quad \text{datasize};
\]

\[
\text{case} \ \text{ftype} \ \text{of}
\]

\[
\quad \text{when} \quad '00' \quad \text{datasize} = 32;
\]

\[
\quad \text{when} \quad '01' \quad \text{datasize} = 64;
\]

\[
\quad \text{when} \quad '1x' \quad \text{UNDEFINED;}
\]

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.
- \(<\text{Sd}>\) 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

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\text{bits(datasize) result;}
\]

\[
\text{bits(datasize) operand} = \text{V}[n];
\]

\[
\text{result} = \text{FPRoundIntN}(\text{operand, FPCR, FPRounding\_ZERO, 64});
\]

\[
\text{V}[d] = \text{result};
\]
**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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | Rn | Rd |

*U* o2  o1

**Half-precision**

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: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  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | Rn | Rd |

*U* o2  o1
Single-precision and double-precision

FRINTA \(<Vd>,\langle T\rangle, \langle Vn\rangle,\langle T\rangle\)

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if \(sz:Q = '10'\) then UNDEFINED;
integer \(esize = 32 \ll \text{UInt}(sz)\);
integer \(datasize = \text{if} \; Q = '1' \; \text{then} \; 128 \; \text{else} \; 64\);
integer \(elements = datasize \div esize\);

boolean \(exact = \text{FALSE}\);
FPRounding \(\text{rounding}\);
case \(U:o1:o2\) of
  \begin{align*}
  &\text{when} \; '0xx' \; \text{rounding} = \text{FPDecodeRounding}(o1:o2)\;; \\
  &\text{when} \; '100' \; \text{rounding} = \text{FPRounding_TIEAWAY}\;; \\
  &\text{when} \; '101' \; \text{UNDEFINED}\;; \\
  &\text{when} \; '110' \; \text{rounding} = \text{FPRoundingMode}(FPCR)\;; \; \text{exact} = \text{TRUE}\;; \\
  &\text{when} \; '111' \; \text{rounding} = \text{FPRoundingMode}(FPCR)\;;
  \end{align*}

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":

\[
\begin{array}{c|c}
\text{Q} & \langle T\rangle \\
\hline
0 & 4H \\
1 & 8H
\end{array}
\]

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\[
\begin{array}{c|c|c}
\text{sz} & \text{Q} & \langle T\rangle \\
\hline
0 & 0 & 2S \\
0 & 1 & 4S \\
1 & 0 & \text{RESERVED} \\
1 & 1 & 2D
\end{array}
\]

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\text{CheckFPAdvSIMDEnabled64}();
\text{bits}(datasize) \text{operand} = \text{V}[n];
\text{bits}(datasize) \text{result};
\text{bits}(esize) \text{element};

\text{for e = 0 to elements-1}
  \text{element} = \text{Elem}[\text{operand}, \; e, \; esize];
  \text{Elem}[\text{result}, \; e, \; esize] = \text{FPRoundInt}(\text{element}, \; FPCR, \; \text{rounding}, \; \text{exact});
\text{V}[d] = \text{result};

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
<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>0</td>
<td>ftype</td>
<td>1</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>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>
</tr>
</tbody>
</table>
```

rmode

Half-precision (ftype == 11)
(Armv8.2)

FRINTA <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTA <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTA <Dd>, <Dn>

```plaintext
der = UInt(Rd);
n = UInt(Rn);

datasize = /
    case ftype of
        when '00' datasize = 32;
        when '01' datasize = 64;
        when '10' UNDEFINED;
        when '11'
            if HaveFP16Ext() then
                datasize = 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();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, FPRounding_TIEAWAY, FALSE);

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 \textit{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 \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}

(\texttt{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 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | Rn | Rd |
|-----------------------------------------------|-----------------------------------------------|
| U | 02 | o1 |
\end{verbatim}

\textbf{Half-precision}

\texttt{FRINTI} <\texttt{Vd}.<\texttt{T}>, <\texttt{Vn}.<\texttt{T}>

\begin{verbatim}
if !\texttt{HaveFP16Ext}() then UNDEFINED;

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
\texttt{FPRounding} rounding;
\text{case } U:01:02 of
  \text{when } '0xx' \text{ rounding = } \texttt{FPDecodeRounding}(01:02);
  \text{when } '100' \text{ rounding = } \texttt{FPRounding_TIEAWAY};
  \text{when } '101' \texttt{ UNDEFINED};
  \text{when } '110' \text{ rounding = } \texttt{FPRoundingMode}(\texttt{FPCR}); exact = TRUE;
  \text{when } '111' \text{ rounding = } \texttt{FPRoundingMode}(\texttt{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 | Q | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rn | Rd |
|-----------------------------------------------|-----------------------------------------------|
| U | 02 | o1 |
\end{verbatim}
Single-precision and double-precision

FRINTI \(<Vd>\),\(<T>\), \(<Vn>\)\(<T>\)

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if \(sz:Q == '10'\) then UNDEFINED;
integer \(esize = 32 \ll \text{UInt}(sz)\);
integer \(datasize = \text{if } Q == '1' \text{ then } 128 \text{ else } 64\);
integer \(elements = datasize \div esize\);

boolean \(\text{exact} = \text{FALSE}\);
\(\text{FPRounding} \text{ rounding}\);

\begin{verbatim}
case U:o1:o2 of
  when '0xx' \(\text{rounding} = \text{FPDecodeRounding}(o1:o2)\);
  when '100' \(\text{rounding} = \text{FPRounding_TIEAWAY}\);
  when '101' UNDEFINED;
  when '110' \(\text{rounding} = \text{FPRoundingMode}(\text{FPCR})\); \(\text{exact} = \text{TRUE}\);
  when '111' \(\text{rounding} = \text{FPRoundingMode}(\text{FPCR})\);
\end{verbatim}

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{tabular}{c|c}
  \(Q\) & \(<T>\) \\
  \hline
  0 & 4H \\
  1 & 8H \\
\end{tabular}

For the single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

\begin{tabular}{c|c|c}
  \(sz\) & \(Q\) & \(<T>\) \\
  \hline
  0 & 0 & 2S \\
  0 & 1 & 4S \\
  1 & 0 & \text{RESERVED} \\
  1 & 1 & 2D \\
\end{tabular}

\(<Vn>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = \(V[n]\);
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
  element = \text{Elem}[operand, e, esize];
  \text{Elem}[result, e, esize] = \text{FPRoundInt}(element, \text{FPCR}, \text{rounding}, \text{exact});
\end{verbatim}

\(V[d] = \text{result};\)
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | Rd |

Half-precision (ftype == 11)
(Armv8.2)

FRINTI <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTI <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTI <Dd>, <Dn>

```plaintext
integer d = UInt(Rd);
integer n = UInt(Rn);

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;
    end
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

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = 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)**

<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>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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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**

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;
FRounding 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

<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>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>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></td>
</tr>
<tr>
<td>U</td>
<td>o2</td>
<td>o1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

FRINTM (vector)
Single-precision and double-precision

FRINTM <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 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;
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 0 1 1 1 1 0 ftype 1 0 0 1 0 1 0 1 0 0 0 0 Rn Rd

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 datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11' if HaveFP16Ext() then
datasize = 16;
else
  UNDEFINED;
FPRounding rounding;
rounding = FPDecodeRounding('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

CheckFPAdvSIMEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = 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)

<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>U</td>
</tr>
</tbody>
</table>

Half-precision

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;
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

<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>U</td>
</tr>
</tbody>
</table>
**Single-precision and double-precision**

FRINTN `<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 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><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 single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>sz</th>
<th>Q</th>
<th><code>&lt;T&gt;</code></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.

| 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  | 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>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;

case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
datassize = 16;
    else
      UNDEFINED;
    FPRounding rounding;
    rounding = FPDencodeRounding('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();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, 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 0 1 1 1 0 1 1 1 1 0 0 1 1 0 0 0 1 0 0 1 0 | Rn | Rd |
| U o2 o1                                   |   |    |

Half-precision

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;
FRounding 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 1 1 1 0 1 sz 1 0 0 0 0 1 1 0 0 0 1 0 | Rn | Rd |
| U o2 o1                                   |   |    |
Single-precision and double-precision

FRINTP <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;
in 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();
bite(datasize) operand = V[n];
bite(datasize) result;
bite(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

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

Half-precision (ftype == 11) (Armv8.2)

FRINTP <Hd>, <Hn>

Single-precision (ftype == 00)

FRINTP <Sd>, <Sn>

Double-precision (ftype == 01)

FRINTP <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

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;

FP Rounding 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();

bits(datasize) result;
bits(datasize) operand = V[n];

result = 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 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 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 Q 1 0 1 1 1 0 0 1 1 0 0 1 1 0 Rn Rd</td>
</tr>
<tr>
<td>U o2 o1</td>
</tr>
</tbody>
</table>

Half-precision

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: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

<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 0 0 1 1 0 0 1 1 0 Rn Rd</td>
</tr>
<tr>
<td>U o2 o1</td>
</tr>
</tbody>
</table>
Single-precision and double-precision

**FRINTX** `<Vd>,<T>, <Vn>,<T>`

```plaintext
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**

```plaintext
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

```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 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 datasize;
case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11'
    if HaveFP16Ext() then
      datasize = 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();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, TRUE);

V[d] = 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 normal for 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   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| U   | o2  | o1  | Rn  | Rd  |
```

### Half-precision

**FRINTZ** `<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   |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| U   | o2  | o1  | Rn  | Rd  |
```
Single-precision and double-precision

FRINTZ \(<Vd>,<T>, <Vn>,<T>\)

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if \(sz:Q == '10'\) then UNDEFINED;
integer \(esize = 32 \ll \text{UInt}(sz)\);
integer \(datasize = \text{if } Q == '1' \text{ then } 128 \text{ else } 64\);
integer \(elements = datasize \div esize\);

boolean \(exact = \text{FALSE}\);
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' \(rounding = \text{FPRoundingMode}(\text{FPCR})\);

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 = \(\text{Elem}[\text{operand}, e, esize]\);
  \(\text{Elem}[\text{result}, e, esize] = \text{FPRoundInt}(\text{element, FPCR, rounding, exact})\);

\(V[d] = \text{result}\);
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 1 0 ftype 1 0 0 1 0 1 1 1 1 0 0 0 0 Rn Rd

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 datasize;

case ftype of
  when '00' datasize = 32;
  when '01' datasize = 64;
  when '10' UNDEFINED;
  when '11' if HaveFP16Ext() then
    datasize = 16;
  else
    UNDEFINED;

FPRounding rounding;
rounding = FPDcodeRounding('11');

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();

bits(datasize) result;
bits(datasize) operand = V[n];

result = FPRoundInt(operand, FPCR, rounding, FALSE);

V[d] = result;
FRSQRTE

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)

```
31 30 29 28 27 26 25 24 23 22 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 0 1 0 1 1 0 0 1 1 0 1 1 0 0 1 1 1 0 1 1 0
Rn Rd
```

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 0 0 0 0 0 1 1 1 0 0 1 1 1 1 0 0 1 1 0
Rn Rd
```

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 1 1 1 0 1 1 1 1 0 0 1 1 1 0 1 1 1 0 0 1 1 0
Rn Rd
```
Vector half precision

FSRQRT\(e <V_d>,<T>,<V_n>,<T>\)

if \(!\text{HaveFP16Ext}()\) then UNDEFINED;

integer \(d = \text{UInt}(\text{Rd})\);
integer \(n = \text{UInt}(\text{Rn})\);

integer \(esize = 16\);
integer \(\text{datasize} = \text{if } Q == '1' \text{ then } 128 \text{ else } 64\);
integer \(\text{elements} = \text{datasize} \div \text{esize}\);

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>
<th>(Rn)</th>
<th>(Rd)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Assembler Symbols

\(<\text{Hd}>\) 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.

\(<\text{V}>\) Is a width specifier, encoded in "sz":

<table>
<thead>
<tr>
<th>(\text{sz})</th>
<th>(\text{&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>

\(<\text{d}>\) Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{n}>\) Is the number of the SIMD&FP source register, encoded in the "Rn" field.

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{T}>\) For the vector half precision variant: is an arrangement specifier, encoded in "\(Q\)":

<table>
<thead>
<tr>
<th>(\text{Q})</th>
<th>(\text{&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 vector single-precision and double-precision variant: is an arrangement specifier, encoded in "sz:Q":

<table>
<thead>
<tr>
<th>(\text{sz})</th>
<th>(\text{Q})</th>
<th>(\text{&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

\( \text{CheckFPAdvSIMDEnabled64}() \);

bits(datasize) operand = \( V[n] \);

bits(datasize) result;

bits(esize) element;

\begin{verbatim}
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSwrtEstimate(element, FPCR);
\end{verbatim}

\( V[d] = \text{result} \);
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 |

### 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 sz 1   |   Rm           |   1 1 1 1 1 1   |   Rd |

### 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 0   |   Rm           |   0 0 1 1 1 1   |   Rd |
Vector half precision

FRSQRTS <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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 | sz | 1 | Rm | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Vector single-precision and double-precision

FRSQRTS <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>&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 vector 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 vector 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] = 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)**

![Half-precision encoding](image)

**Single-precision and double-precision**

![Single-precision and double-precision encoding](image)

**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;<strong>T</strong>&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":

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;
<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];
    Elem[result, e, esize] = FPSqrt(element, FPCR);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 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 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    | ftype |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | opc |

Half-precision (ftype == 11)  (Armv8.2)

FSQRT <Hd>, <Hn>

Single-precision (ftype == 00)

FSQRT <Sd>, <Sn>

Double-precision (ftype == 01)

FSQRT <Dd>, <Dn>

integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize;

<table>
<thead>
<tr>
<th>case ftype of</th>
</tr>
</thead>
<tbody>
<tr>
<td>when '00' datasize = 32;</td>
</tr>
<tr>
<td>when '01' datasize = 64;</td>
</tr>
<tr>
<td>when '10' UNDEFINED;</td>
</tr>
<tr>
<td>when '11'</td>
</tr>
<tr>
<td>if HaveFP16Ext() then</td>
</tr>
<tr>
<td>datasize = 16;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>UNDEFINED;</td>
</tr>
</tbody>
</table>

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

```c
CheckFPAdvSIMDEnabled64();

bits(datasize) result;
bits(datasize) operand = V[n];

result = 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)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | Rm | 0  | 0  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  |

#### Half-precision

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | Rm | 1  | 1  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  |

#### Single-precision and double-precision

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>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4H</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)
<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;
bits(esize) diff;
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | ftype | 1 | Rm | 0 | 0 | 1 | 1 | 1 | 0 | Rn | Rd | op

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>

t = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);

type

case ftype of
  | when '00' datasize = 32;
  | when '01' datasize = 64;
  | when '10' UNDEFINED;
  | when '11'
    | if HaveFP16Ext() then
      | datasize = 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

`CheckFPAdvSIMEnabled64();`
`bits(datasize) result;`
`bits(datasize) operand1 = V[n];`
`bits(datasize) operand2 = V[m];`

result = FSub(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)`.

---

**Advanced SIMD**

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 idxdsize = 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><code>imm5</code></th>
<th><code>&lt;Ts&gt;</code></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><code>imm5</code></th>
<th><code>&lt;index1&gt;</code></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><code>imm5</code></th>
<th><code>&lt;index2&gt;</code></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

```
CheckFPAdvSIMDEnabled64();
bits(idxds) operand = V[n];
bits(128) result;

result = V[d];
Elem[result, dst_index, esize] = Elem[operand, src_index, 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.
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 CPACR_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 (from general).


<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 0 0 0 0</td>
</tr>
</tbody>
</table>

Advanced SIMD

INS <Vd>.<Ts>[<index>], <R><n>

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 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 &lt;Ts&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>xxx1</td>
</tr>
<tr>
<td>xxx10</td>
</tr>
<tr>
<td>xx100</td>
</tr>
<tr>
<td>x1000</td>
</tr>
</tbody>
</table>

<index> Is the element index encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5 &lt;index&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>xxx1</td>
</tr>
<tr>
<td>xxx10</td>
</tr>
<tr>
<td>xx100</td>
</tr>
<tr>
<td>x1000</td>
</tr>
</tbody>
</table>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

<table>
<thead>
<tr>
<th>imm5 &lt;R&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>xxx1</td>
</tr>
<tr>
<td>xxx10</td>
</tr>
<tr>
<td>xx100</td>
</tr>
<tr>
<td>x1000</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

```c
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n];
bits(128) result;

result = Y[d];
Elem[result, index, 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.
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 CPACR_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 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | x | x | 1 | x | size | Rn | Rt |

L  
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 = 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 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | x | x | 1 | x | size | Rn | Rt |

L  
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>

```plaintext
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

```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 (3 registers)
  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
  SetNotTagCheckedInstruction(!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 \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: \textbf{No offset} and \textbf{Post-index}

\textbf{No offset}

\begin{verbatim}
<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>x</th>
<th>x</th>
<th>0</th>
<th>S</th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>R</td>
<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>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{8-bit (opcode == 000)}

LD1 \{ <Vt>.B \}[<index>], [<Xn|SP>]

\textbf{16-bit (opcode == 010 \&\& size == x0)}

LD1 \{ <Vt>.H \}[<index>], [<Xn|SP>]

\textbf{32-bit (opcode == 100 \&\& size == 00)}

LD1 \{ <Vt>.S \}[<index>], [<Xn|SP>]

\textbf{64-bit (opcode == 100 \&\& S == 0 \&\& size == 01)}

LD1 \{ <Vt>.D \}[<index>], [<Xn|SP>]

integer t = \texttt{UInt}(Rt);
integer n = \texttt{UInt}(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

\textbf{Post-index}

\begin{verbatim}
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rm | x | x | 0 | S | size | Rn | Rt |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| L | R | opcode |
\end{verbatim}
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 scale = UInt(opcode<2:1>);
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
  SetNotTagCheckedInstruction(!tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bids(64) offs;
bids(128) rval;
bids(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.
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 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | size | Rn | Rt |

L R | opcode | S

No offset

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 | 0 | 0 | size | Rm | 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”:

<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>

\(<Xn|SP>\) 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 “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#1</td>
</tr>
<tr>
<td>01</td>
<td>#2</td>
</tr>
<tr>
<td>10</td>
<td>#4</td>
</tr>
<tr>
<td>11</td>
<td>#8</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
text
```

integer scale = UInt(opcode<2:1>);
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 \( \text{HaveMTEExt}() \) then
   \( \text{SetNotTagCheckedInstruction(!tag\_checked);} \)

\( \text{CheckFPAdvSIMDEnabled64();} \)

\[
\begin{align*}
\text{bits(64) address; } \\
\text{bits(64) offs; } \\
\text{bits(128) rval; } \\
\text{bits(esize) element; } \\
\text{constant integer ebytes = esize DIV 8; }
\end{align*}
\]

if \( n == 31 \) then
   \( \text{CheckSPAlignment();} \)
   \( \text{address = SP[];} \)
else
   \( \text{address = X[n];} \)
offs = \( \text{Zeros();} \)
if replicate then
   // load and replicate to all elements
   for \( s = 0 \) to \( \text{selem}-1 \)
      element = \( \text{Mem[address+offs, ebytes, AccType\_VEC];} \)
   // replicate to fill 128- or 64-bit register
   \( \text{V[t]} = \text{Replicate(element, datasize DIV esize);} \)
   \( \text{offs = offs + ebytes;} \)
   \( t = (t + 1) \mod 32; \)
else
   // load/store one element per register
   for \( s = 0 \) to \( \text{selem}-1 \)
      rval = \( \text{V[t]}; \)
   if memop == \( \text{MemOp\_LOAD} \) then
      // insert into one lane of 128-bit register
      \( \text{Elem[rval, index, esize]} = \text{Mem[address+offs, ebytes, AccType\_VEC];} \)
      \( \text{V[t]} = \text{rval;} \)
   else // memop == \( \text{MemOp\_STORE} \)
      // extract from one lane of 128-bit register
      \( \text{Mem[address+offs, ebytes, AccType\_VEC]} = \text{Elem[rval, index, esize];} \)
   \( \text{offs = offs + ebytes;} \)
   \( t = (t + 1) \mod 32; \)
if wback then
   if \( m \neq 31 \) then
      \( \text{offs = X[m];} \)
   if \( n == 31 \) then
      \( \text{SP[]} = \text{address + offs;} \)
   else
      \( \text{X[n]} = \text{address + offs;} \)

**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></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD2 {<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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>size</th>
<th>Rm</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>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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

Immediate offset (Rm == 11111)

LD2 {<Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD2 {<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>RESERVED</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2D</td>
</tr>
</tbody>
</table>
Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 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>#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**

```cpp
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
    SetNotTagCheckedInstruction(!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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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

![0x0 encoding](image)

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>]

```java
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = false;
boolean tag_checked = wback || n != 31;
```

### Post-index

![0x0 encoding](image)

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>]

```java
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)
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 == 00)
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)
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 scale = UInt(opcode<2:1>);
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
    SetNotTagCheckedInstruction(!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();
adress = 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

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>opcode</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Post-index

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>opcode</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Immediate offset (Rm == 11111)**

LD2R { <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>], [imm]

**Register offset (Rm != 11111)**

LD2R { <Vt>..<T>, <Vt2>..<T> }, [<Xn|SP>], <Xm>

### 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>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>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”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;imm&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>#2</td>
</tr>
<tr>
<td>01</td>
<td>#4</td>
</tr>
<tr>
<td>10</td>
<td>#8</td>
</tr>
<tr>
<td>11</td>
<td>#16</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
text
integer scale = UInt(opcode<2:1>);
text
integer selem = UInt(opcode<0>:R) + 1;
text
boolean replicate = FALSE;
text
integer index;
text
case scale of
  when 3
    // load and replicate
    if L == '0' || S == '1' then UNDEFINED;
text
    scale = UInt(size);
text
    replicate = TRUE;
text
  when 0
    index = UInt(Q:S:size);  // B[0-15]
text
  when 1
    if size<0> == '1' then UNDEFINED;
text
    index = UInt(Q:S:size<1>);  // H[0-7]
text
  when 2
    if size<1> == '1' then UNDEFINED;
text
    if size<0> == '0' then
      index = UInt(Q:S);  // S[0-3]
text
    else
      if S == '1' then UNDEFINED;
text
      index = UInt(Q);  // D[0-1]
text
    scale = 3;
text

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
text
integer datasize = if Q == '1' then 128 else 64;
text
integer esize = 8 << scale;
```
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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];
        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

Immediate offset (Rm == 11111)

Register offset (Rm != 11111)
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>&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

```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;
```

LD3 (multiple structures)
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!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 \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{No offset} and \textit{Post-index}

\textbf{No offset}

\begin{center}
\begin{tabular}{cccccccccccccccc}
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & 1 & S & \text{size} & Rn & Rt \\
\end{tabular}
\end{center}

\textbf{8-bit (opcode == 001)}

LD3 \{ <Vt>.B, <Vt2>.B, <Vt3>.B \}[<index>], [<Xn|SP>]

\textbf{16-bit (opcode == 011 && size == x0)}

LD3 \{ <Vt>.H, <Vt2>.H, <Vt3>.H \}[<index>], [<Xn|SP>]

\textbf{32-bit (opcode == 101 && size == 00)}

LD3 \{ <Vt>.S, <Vt2>.S, <Vt3>.S \}[<index>], [<Xn|SP>]

\textbf{64-bit (opcode == 101 && S == 0 && size == 01)}

LD3 \{ <Vt>.D, <Vt2>.D, <Vt3>.D \}[<index>], [<Xn|SP>]

\begin{verbatim}
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
\end{verbatim}

\textbf{Post-index}

\begin{center}
\begin{tabular}{cccccccccccccccc}
\hline
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & x & x & 1 & S & \text{size} & Rn & Rt \\
\end{tabular}
\end{center}
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 scale = UInt(opcode<2:1>);
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
    SetNotTagCheckedInstruction(!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
    CheckSPA.Align();
    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 \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 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| L | R | Ope | code | S |

**No offset**

LD3R \{ \texttt{<Vt>..<T>}, \texttt{<Vt2>..<T>}, \texttt{<Vt3>..<T>} \}, \{\texttt{<Xn|SP>}\}

integer \( t = \texttt{UInt}(Rt) \);
integer \( n = \texttt{UInt}(Rn) \);
integer \( m = \texttt{integer\_UNKNOWN} \);
boolean \( wback = \texttt{FALSE} \);
boolean \( \text{tag\_checked} = wback || n != 31 \);

**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 | 1 | 1 | 0 | 1 | 1 | 0 | size | Rm | 1 | 1 | 1 | 0 | size | Rn | Rt |
| L | R | Ope | code | S |

**Immediate offset (Rm == 11111)**

LD3R \{ \texttt{<Vt>..<T>}, \texttt{<Vt2>..<T>}, \texttt{<Vt3>..<T>} \}, \{\texttt{<Xn|SP>}\}, \texttt{<imm>}

**Register offset (Rm != 11111)**

LD3R \{ \texttt{<Vt>..<T>}, \texttt{<Vt2>..<T>}, \texttt{<Vt3>..<T>} \}, \{\texttt{<Xn|SP>}\}, \texttt{<Xm>}

integer \( t = \texttt{UInt}(Rt) \);
integer \( n = \texttt{UInt}(Rn) \);
integer \( m = \texttt{UInt}(Rm) \);
boolean \( wback = \texttt{TRUE} \);
boolean \( \text{tag\_checked} = wback || n != 31 \);

**Assembler Symbols**

\texttt{<Vt>} \quad Is the name of the first or only SIMD&FP register to be transferred, encoded in the “Rt” field.

\texttt{<T>} \quad Is an arrangement specifier, encoded in “size:Q”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>\texttt{&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>

\texttt{<Vt2>} \quad Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 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>#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>

Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the “Rm” field.

Shared Decode

```plaintext
text
integer scale = UInt(opcode<2:1>);
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 \( \text{HaveMTEExt}() \) then
  SetNotTagCheckedInstruction(!tag_checked);

\textbf{CheckFPAdvSIMDEnabled64}();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(128) element;
constant integer ebytes = esize DIV 8;

if \( n = 31 \) then
  \textbf{CheckSPAlignment}();
  address = SP[];
else
  address = X[n];

offs = \textbf{Zeros}();
if replicate then
  // load and replicate to all elements
  for s = 0 to selem-1
    element = \textbf{Mem}[address+offs, ebytes, AccType_VEC];
    \textbf{V}[t] = \textbf{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 = \textbf{V}[t];
    if memop == \textbf{MemOp_LOAD} then
      \textbf{Elem}[rval, index, esize] = \textbf{Mem}[address+offs, ebytes, AccType_VEC];
    else // memop == \textbf{MemOp_STORE}
      \textbf{Mem}[address+offs, ebytes, AccType_VEC] = \textbf{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;

\textbf{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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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
  |   Q 0  0  1  1  0  0  0  1  0  0  0  0  0  0  0  0 | size  Rn  Rt
  L               opcode
```

No offset

```
LD4 { <Vt>.,<T>, <Vt2>.,<T>, <Vt3>.,<T>, <Vt4>.,<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

```
  31  30  29  28  27  26  25  24  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  0  1  1  0 | Rm  0  0  0  0 | size  Rn  Rt
  L               opcode
```

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>
```

```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>
```
Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

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**

Let:

- \( L \) = if \( L = '1' \) then MemOp LOAD else MemOp STORE;
- \( Q \) = if \( Q = '1' \) then 128 else 64;
- \( size = 8 \ll \text{UInt}(size) \);
- \( esize = \text{datasize} \div \text{esize} \);
- \( \text{elements} = \text{datasize} \div \text{esize} \);
- \( \text{rpt} \); // number of iterations
- \( \text{selem} \); // structure elements

\begin{verbatim}
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 (3 registers)
  when '1000' rpt = 1; selem = 2; // LD/ST2 (2 registers)
  when '1010' rpt = 2; selem = 1; // LD/ST1 (2 registers)
  otherwise UNDEFINED;
endcase

// .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
    SetNotTagCheckedInstruction(!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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | x  | x  | 1  | S  | size | Rn | Rr |
| 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);
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  | Rm | x  | x  | 1  | S  | size | Rn | Rr |
| 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 scale = UInt(opcode<2:1>);
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
  SetNotTagCheckedInstruction(!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

No offset


integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;

Post-index

Post-index

Immediate offset (Rm == 11111)


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.
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 “size”:

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

```c
integer scale = UInt(opcode<2:1>);
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 \texttt{HaveMTEExt}() then
    \texttt{SetNotTagCheckedInstruction(!tag\_checked)};

\texttt{CheckFPAdvSIMDEnabled64}();

\par

bits(64) \texttt{address};
bits(64) \texttt{offs};
bits(128) \texttt{rval};
bits(128) \texttt{element};
constant integer \texttt{ebytes} = \texttt{esize} \texttt{DIV} 8;

if \texttt{n == 31} then
    \texttt{CheckSPAlignment}();
    \texttt{address = \texttt{SP}[]};
else
    \texttt{address} = \texttt{X[n]};

\texttt{offs} = \texttt{Zeros}();

if \texttt{replicate} then
    // load and replicate to all elements
    for \texttt{s = 0} to \texttt{selem-1}
        \texttt{element} = \texttt{Mem[address+offs, ebytes, AccType\_VEC]};
        // replicate to fill 128- or 64-bit register
        \texttt{V[t]} = \texttt{Replicate(element, datasize \texttt{DIV} esize)};
        \texttt{offs} = \texttt{offs + ebytes};
        \texttt{t} = (t + 1) \texttt{MOD} 32;
else
    // load/store one element per register
    for \texttt{s = 0} to \texttt{selem-1}
        \texttt{rval} = \texttt{V[t]};
        if \texttt{memop == MemOp\_LOAD} then
            // insert into one lane of 128-bit register
            \texttt{Elem[rval, index, esize]} = \texttt{Mem[address+offs, ebytes, AccType\_VEC]};
            \texttt{V[t]} = \texttt{rval};
        else // memop == MemOp\_STORE
            // extract from one lane of 128-bit register
            \texttt{Mem[address+offs, ebytes, AccType\_VEC]} = \texttt{Elem[rval, index, esize]};
            \texttt{offs} = \texttt{offs + ebytes};
            \texttt{t} = (t + 1) \texttt{MOD} 32;

if \texttt{wback} then
    if \texttt{m != 31} then
        \texttt{offs} = \texttt{X[m]};
    if \texttt{n == 31} then
        \texttt{SP[]} = \texttt{address} + \texttt{offs};
    else
        \texttt{X[n]} = \texttt{address} + \texttt{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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | imm7| Rt2 | Rn  | Rt  |
\[L\]

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> 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

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;
bcr(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
Operation

```plaintext
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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();

if n == 31 then
    CheckSPAlignment();
    address = SP[];
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>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></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>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></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>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

boolean wback = TRUE;
boolean postindex = TRUE;


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;
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;
boolean rt_unknown = FALSE;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!tag_checked);

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();

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>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>size</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</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>

**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>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>size</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>1</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>
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;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| size | 1 | 1 | 1 | 1 | 0 | 1 | x | 1 | imm12 | Rn | Rt |

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;
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.
- `<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 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);
```

LDR (immediate, SIMD&FP)
if \( \text{HaveMTEExt}() \) then
    \( \text{SetNotTagCheckedInstruction(!tag\_checked);} \)

\( \text{CheckFPAdvSIMDEnabled64}(); \)

\( \text{bits(64) address;} \)

\( \text{bits(datasize) data;} \)

if \( n == 31 \) then
    \( \text{CheckSPAlignment();} \)
    \( \text{address} = \text{SP[];} \)
else
    \( \text{address} = \text{X[n];} \)

if \( \text{!postindex} \) then
    \( \text{address} = \text{address + offset;} \)

\( \text{case memop of} \)

    \( \text{when MemOp\_STORE} \)
    \( \text{data} = V[t]; \)
    \( \text{Mem}[\text{address, datasize DIV 8, AccType\_VEC}] = data; \)

    \( \text{when MemOp\_LOAD} \)
    \( \text{data} = \text{Mem}[\text{address, datasize DIV 8, AccType\_VEC}]; \)
    \( V[t] = data; \)

if \( \text{wback} \) then
    if \( \text{postindex} \) then
        \( \text{address} = \text{address + offset;} \)
    if \( n == 31 \) then
        \( \text{SP[]} = \text{address;} \)
    else
        \( \text{X[n]} = \text{address;} \)

\textbf{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 | Rt |

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
  SetNotTagCheckedInstruction(TRUE);
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | x  | 1  | 1  | Rm | option | S | 1 | 0 | Rn | Rm | | | | | | | | | | | | | | |

8-fsreg,LDR-8-fsreg (size == 00 & opc == 01 & option != 011)

LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]  

8-fsreg,LDR-8-fsreg (size == 00 & opc == 01 & option == 011)

LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]  

16-fsreg,LDR-16-fsreg (size == 01 & opc == 01)

LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

32-fsreg,LDR-32-fsreg (size == 10 & opc == 01)

LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){{, <extend> {<amount>}}}

64-fsreg,LDR-64-fsreg (size == 11 & opc == 01)

LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){{, <extend> {<amount>}}}

128-fsreg,LDR-128-fsreg (size == 00 & opc == 11)

LDR <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 = \texttt{ExtendReg}(m, \text{extend\_type}, \text{shift});
if \texttt{HaveMTEExt}() then
  \texttt{SetNotTagCheckedInstruction(!tag\_checked)};
\texttt{CheckFPAdvSIMDEnabled64}();
bits(64) address;
bits(datasize) data;
if n == 31 then
  \texttt{CheckSPAlignment}();
  address = SP[];
else
  address = X[n];
address = address + offset;
case memop of
  when \texttt{MemOp\_STORE}
    data = V[t];
    \texttt{Mem}[address, \text{datasize DIV 8, AccType\_VEC}] = data;
  when \texttt{MemOp\_LOAD}
    data = \texttt{Mem}[address, \text{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.

```
size  1  1  1  1  0  0  x  1  0   imm9       0  0   Rn    Rt 

opc  
```

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.

< 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

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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.
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 CPACR_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 | 1 | 0 | 1 | 1 | 1 | size | L | M | Rm | 0 | 0 | 0 | 0 | H | 0 | Rn | Rd |
 02
```

Vector

MLA <Vd>.<T>, <Vn>.<T>, <Vn>.<Vm>[<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 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”:

```
<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
Operation

```CheckFPAdvSIMDEnabled64();```
bits(datasize) operand1 = V[n];
bits(idxsiz) 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-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 \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.

Three registers of the same type

\text{MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

\begin{verbatim}
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');
\end{verbatim}

Assembler Symbols

\textbf{<Vd>} \hspace{1em} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\textbf{<T>} \hspace{1em} 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>

\textbf{<Vn>} \hspace{1em} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\textbf{<Vm>} \hspace{1em} Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

\begin{verbatim}
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;
\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.
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  |  L  |  M  |  Rm  |  0  |  1  |  0  |  0  |  H  |  0  |  Rn  |  Rd
```

Vector

MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<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 = 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”:

```
<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>
```
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) 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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
MLS (vector)

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.

Three registers of the same type

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.
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 \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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | Q  | op | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | a  | b  | c  | cmode | 0  | 1  | d  | e  | f  | g  | h  | Rd |
\end{verbatim}

8-bit (\texttt{op} == 0 & \& \texttt{cmode} == 1110)

\begin{verbatim}
MOVI <Vd>.<T>, #<imm8>{, LSL #0}
\end{verbatim}

16-bit shifted immediate (\texttt{op} == 0 & \& \texttt{cmode} == 10x0)

\begin{verbatim}
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
\end{verbatim}

32-bit shifted immediate (\texttt{op} == 0 & \& \texttt{cmode} == 0xx0)

\begin{verbatim}
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
\end{verbatim}

32-bit shifting ones (\texttt{op} == 0 & \& \texttt{cmode} == 110x)

\begin{verbatim}
MOVI <Vd>.<T>, #<imm8>, MSL #<amount>
\end{verbatim}

64-bit scalar (\texttt{Q} == 0 & \& \texttt{op} == 1 & \& \texttt{cmode} == 1110)

\begin{verbatim}
MOVI <Dd>, #<imm>
\end{verbatim}

64-bit vector (\texttt{Q} == 1 & \& \texttt{op} == 1 & \& \texttt{cmode} == 1110)

\begin{verbatim}
MOVI <Vd>.2D, #<imm>
\end{verbatim}

```c
integer rd = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
basis(datasize) imm;
basis(64) imm64;

ImmediateOp operation;
case cmode:op of
  when '0xx00' operation = ImmediateOp_MOVI;
  when '0xx01' operation = ImmediateOp_MVNI;
  when '0xx10' operation = ImmediateOp_ORR;
  when '0xx11' operation = ImmediateOp_BIC;
  when '10x00' operation = ImmediateOp_MOVI;
  when '10x01' operation = ImmediateOp_MVNI;
  when '10x10' operation = ImmediateOp_ORR;
  when '10x11' operation = ImmediateOp_BIC;
  when '110x0' operation = ImmediateOp_MOVI;
  when '110x1' operation = ImmediateOp_MVNI;
  when '1110x' operation = ImmediateOp_MOVI;
  when '11110' operation = ImmediateOp_MOVI;
  when '11111' 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 "aaaaaaaabbbbbcddddddeeeeeefffffffgggggggghhhhhhh", encoded in "a:b:c:d:e:f:g:h".
<T> For the 8-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>8B</td>
</tr>
<tr>
<td>1</td>
<td>16B</td>
</tr>
</tbody>
</table>

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>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 shifted immediate 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 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

```
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.
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.

Vector

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></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></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>L</td>
</tr>
<tr>
<td>10</td>
<td>H:M</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
### Operation

```plaintext
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.

Three registers of the same type

\[\text{MUL } <V_d>.<T>, <V_n>.<T>, <V_m>.<T>\]

\[
\begin{align*}
\text{integer } d & = \text{UInt}(Rd); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{integer } m & = \text{UInt}(Rm); \\
\text{if } U == '1' \&\& \text{ size } != '00' \text{ then UNDEFINED;} \\
\text{if } \text{ size } == '11' \text{ then UNDEFINED;} \\
\text{integer } \text{ esize } & = 8 << \text{UInt}(\text{size}); \\
\text{integer } \text{ datasize } & = \text{if } Q == '1' \text{ then } 128 \text{ else } 64; \\
\text{integer } \text{ elements } & = \text{datasize} \text{ DIV } \text{esize}; \\
\text{boolean } \text{poly} & = (U == '1');
\end{align*}
\]

Assembler Symbols

\(<V_d>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<T>\) Is an arrangement specifier, encoded in "size:Q":

\[
\begin{array}{ccc}
\text{size} & Q & <T> \\
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & \text{RESERVED}
\end{array}
\]

\(<V_n>\) 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

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = \text{V}[n];
bits(datasize) operand2 = \text{V}[m];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;
for e = 0 to elements-1
    \text{element1} = \text{Elem}[\text{operand1, e, esize}];
    \text{element2} = \text{Elem}[\text{operand2, e, esize}];
    if \text{poly} then
        \text{product} = \text{PolynomialMult}(\text{element1}, \text{element2})\text{<esize-1:0>};
    else
        \text{product} = (\text{UInt}(\text{element1}) \times \text{UInt}(\text{element2}))\text{<esize-1:0>};
    \text{Elem}[\text{result, e, esize}] = \text{product};
\text{V}[d] = \text{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.
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 `CPACR_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 shifted immediate (cmode == 10x0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

#### 32-bit shifted immediate (cmode == 0xx0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

#### 32-bit shifting ones (cmode == 110x)

```
MVNI <Vd>.<T>, #<imm8>, MSL #<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 '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":
  - `Q | <T>`
    - `0 | 4H`
    - `1 | 8H`
  For the 32-bit variant: is an arrangement specifier, encoded in "Q":
  - `Q | <T>`
    - `0 | 25`
    - `1 | 45`
- `<amount>`: For the 16-bit shifted immediate variant: is the shift amount encoded in "cmode<1>".
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>

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.
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | Rn | Rd |

Scalar

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | Rn | Rd |

Vector

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

<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>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>

<i>Vn</i> 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.

Vector

\[
\begin{array}{cccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
\end{array}
\]

\[\text{Rn} \quad \text{Rd}\]

\text{NOT} <Vd>,<T>, <Vn>,<T>

integer \(d = \text{UInt}(\text{Rd})\);
integer \(n = \text{UInt}(\text{Rn})\);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;

Assembler Symbols

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
\(<\text{T}>\) Is an arrangement specifier, encoded in "Q":
\[
\begin{array}{c|c}
Q & <\text{T}> \\
\hline
0 & 8B \\
1 & 16B \\
\end{array}
\]

\(<\text{Vn}>\) Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

\text{CheckFPAdvSIMDEnabled64}();

bits(datasize) operand = \text{V}[n];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
\[
\text{element} = \text{Elem}[\text{operand}, e, \text{esize}];
\text{Elem}[\text{result}, e, \text{esize}] = \text{NOT}(%text{element});
\]
\text{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></th>
<th>Q</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Rm</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>Rd</td>
</tr>
</tbody>
</table>

Three registers of the same type

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.

31 30 29 28 27 26 25 24 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>op</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></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></td>
<td>2S</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>":

ORR (vector, immediate)
<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 \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 instruction is used by the alias \texttt{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 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | Rm | 0 | 0 | 0 | 1 | 1 | 1 | Rn | Rd |
| size |

Three registers of the same type

\texttt{ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

```plaintext
integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

**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 “Q”:
  - \texttt{Q <T>}
    - 0 8B
    - 1 16B
- \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.

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{MOV (vector)}</td>
<td>\texttt{Rm} == \texttt{Rn}</td>
</tr>
</tbody>
</table>

**Operation**

```plaintext
\texttt{CheckFPAdvSIMDEnabled64()};
bits(datasize) operand1 = \texttt{V}[n];
bits(datasize) operand2 = \texttt{V}[m];
bits(datasize) result;
result = operand1 OR operand2;
\texttt{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.
PMUL

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.

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>1</th>
<th>Rm</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three registers of the same type

PMUL <Vd>,<T>, <Vn>,<T>, <Vm>,<T>

```
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>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

```
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.

---

### Three registers, not all the same type

**PMULL\{2\}**  <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

- integer \(d = \text{UInt}(\text{Rd})\);
- integer \(n = \text{UInt}(\text{Rn})\);
- integer \(m = \text{UInt}(\text{Rm})\);

\[
\text{if } \text{size} == '01' \text{ || size} == '10' \text{ then UNDEFINED;}
\]

\[
\text{if } \text{size} == '11' \text{ && !} \text{HaveBit128PMULLExt()} \text{ then UNDEFINED;}
\]

- integer \(\text{esize} = 8 << \text{UInt}(\text{size});\)
- integer \(\text{datasize} = 64;\)
- integer \(\text{part} = \text{UInt}(\text{Q});\)

\[\text{integer } \text{elements} = \frac{\text{datasize}}{\text{DIV}} \text{ esize};\]

---

### Assembler Symbols

2 \[ \text{Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q":} \]

<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> \[ \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.} \]

<Ta> \[ \text{Is an arrangement specifier, encoded in "size":} \]

<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>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

The ‘1Q’ arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> \[ \text{Is the name of the first SIMD&FP source register, encoded in the "Rn" field.} \]

<Tb> \[ \text{Is an arrangement specifier, encoded in "size:Q":} \]

<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>
</tbody>
</table>

<Vm> \[ \text{Is the name of the second SIMD&FP source register, encoded in the "Rm" field.} \]
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part];
bits(datasize) operand2 = Vpart[m, part];
bits(2*datasize) result;
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.
RADDING, RADDHNN2

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 RADDING instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHNN2 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.

```
U 0 1 1 1 0 size 1 Rm 0 1 0 0 0 0 0 0 Rd
```

Three registers, not all the same type

RADDHNN{2} <Vd>,<Tb>, <Vn>,<Ta>, <Vm>.<Ta>

```java
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

<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":

```
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.
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 ARMv8.2-SHA is implemented.

Advanced SIMD
(Armv8.2)

```
1 1 0 0 1 1 1 0 0 1 1 | Rm | 1 0 0 0 1 1 | Rn | Rd
```

Advanced SIMD

RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D

```plaintext
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.

<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 0 0 1 1 0 0 0 0 0 0 1 0 1 1 0 Rn Rd</td>
</tr>
</tbody>
</table>

Vector

RBIT <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;
bits(esize) rev;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    for i = 0 to esize-1
        rev<esize-1-i> = element<i>;
    Elem[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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 CPU_CR_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
\[
\begin{array}{c|cccc|cccc|c}
\text{U} & 0 & 0 & 0 & 1 & 1 & 0 & \text{size} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\text{Rn} & & & & & & & & & & & & & & & & & & & & & & & \text{Rd} \\
\end{array}
\]

Vector

REV16 \(<\text{Vd}>.<\text{T}>, <\text{Vn}>.<\text{T}>\)

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;
endcase

integer containers = datasize DIV container_size;
iconteger elements_per_container = container_size DIV esize;

Assembler Symbols

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\(<\text{T}>\) 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>

\(<\text{Vn}>\) 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;
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.
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.

```
0 0     0 0 0 0 0 0 0 0 0 1 0 0 1 | size    | Rd |
0 1 0 1 1 1 0 | Q 0 | o0 | Rn |
U
```

### Vector

```
REV32 <Vd>.<T>, <Vn>.<T>
```

integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   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;
inger integer element = 0;
inger 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.

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**REV64**

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.

- **<Vd>**<T>, <Vn>.<T>

  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>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.

.. raw:: latex

   \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  Q 0 0 1 1 1 1 0 != 0000  immh 1 0 0 0 1 1 1 1  Rn   Rd
    immh  op
   \end{verbatim}

Vector

RSHRN{2} \<Vd>.<Tb>, <Vn>.<Ta>, \#<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);

if imm == '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”:

\[
\begin{array}{c|c}
Q & 2 \\
\hline
0 & \text{[absent]} \\
1 & \text{[present]} \\
\end{array}
\]

\<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\<Tb> Is an arrangement specifier, encoded in “immh:Q”:

\[
\begin{array}{c|c}
\text{immh} & Q \\
\hline
0000 & \text{SEE Advanced SIMD modified immediate} \\
0001 & 0 8B \\
0011 & 1 16B \\
01x  & 0 4H \\
01x  & 1 8H \\
01xx & 0 2S \\
01xx & 1 4S \\
1xxx & \text{reserved} \\
\end{array}
\]

\<Vn> Is the name of the SIMD&FP source register, encoded in the “Rn” field.

\<Ta> Is an arrangement specifier, encoded in “immh”:

\[
\begin{array}{c|c}
\text{immh} & \text{Ta} \\
\hline
0000 & \text{SEE Advanced SIMD modified immediate} \\
0001 & 8H \\
001x & 4S \\
01xx & 2D \\
1xxx & \text{reserved} \\
\end{array}
\]

RSHRN, RSHRN2
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**

```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.
RSUBHN, RSUBHN2

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.

```
0 1 1 1 1 0 | size 1 | Rd
0 1 1 0 0 0 | Rn
U
```

Three registers, not all the same type

RSUBHN(2) <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<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":

```plaintext
<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>01</td>
<td>16B</td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
</tr>
<tr>
<td>10</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>x</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

\texttt{CheckFPAdvSIMEnabled64()};
\texttt{bits(2*datasize) operand1 = V[n];}
\texttt{bits(2*datasize) operand2 = 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 = \textit{Elem}[operand1, e, 2*esize];}
\texttt{ element2 = \textit{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{ \textit{Elem}[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part] = 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}

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SABA

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.

Three registers of the same type

```
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;
```
Operational information

If PSTATE.DIT is 1:

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

### Three registers, not all the same type

SABAL(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

<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><code>&lt;Ta&gt;</code></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><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>

`<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.
SABD

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.

```
<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>Q</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
<td>1</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>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>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Three registers of the same type

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>&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.
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>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>size</th>
<th>1</th>
<th>Rm</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>
</tr>
</tbody>
</table>
```

**Three registers, not all the same type**

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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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
CheckFPAdSiMDEnabled64();
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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

Vector

SADALP <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>&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

CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
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];
    Elem[result, e, 2*esize] = 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.
SADDL, SADDL2

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.

<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>

**Three registers, not all the same type**

SADDL(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**

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>0</th>
<th>[absent]</th>
</tr>
</thead>
<tbody>
<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;
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.
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.

```
<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>Q</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>size</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>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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
```

**Vector**

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>&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();
bits(datasize) operand = V[n];
bits(datasize) result;
bits(2*esize) sum;
integer op1;
integer op2;
result = if acc then V[d] else Zeros();
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>;
    Elem[result, e, 2*esize] = 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.
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.

```
0 0 0 1 1 0 | size 1 1 0 0 0 0 1 1 1 0 | Rd | Rn
```

Advanced SIMD

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”:

```
size Q <T>
00 0 8B
00 1 16B
01 0 4H
01 1 8H
10 0 RESERVED
10 1 4S
11 x RESERVED
```

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.

| 31 30 29 28 27 26 25 24 23 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 | 0 | 1 | 0 | 0 | Rn | 0 | 0 | 0 | 0 | 0 | Rd | 0 | 1 |

Three registers, not all the same type

SADDW(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>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

```
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.
SCVT (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 | immb | 1 1 1 0 0 1 | Rn | Rd |
| U | immh |

Scalar

SCVT <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 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 0 1 1 1 1 0 !|= 0000 | immb | 1 1 1 0 0 1 | Rn | Rd |
| U | immh |

Vector

SCVT <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 = FPRoundingMode(FPCR);

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:
<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>001x</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01xx</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>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>01xx</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) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, FPCR, rounding);
\( V[d] \) = result;
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)

31 30 29 28 27 26 25 24 23 22 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 1 1 0 0 1 1 1 0 1 1 0 Rn | Rd
U

Scalar half precision

SCVTF <Hd>, <Hn>

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 0 1 1 1 1 0 0 sz 1 0 0 0 0 1 1 1 0 1 1 0 Rn | Rd
U

Scalar single-precision and double-precision

SCVTF <V>d, <V>n

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 0 1 1 1 0 0 1 1 1 1 0 0 1 1 1 0 1 1 0 Rn | Rd
U
**Vector half precision**

SCVTNF \(<Vd>.<T>, <Vn>.<T>\)

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**

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>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 vector 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 vector 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

\begin{verbatim}
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;
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;
\end{verbatim}

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.
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;
else
  UNDEFINED;
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();

bits(fltsize) fltval;
bits(intsize) intval;
intval = X[n];
fltval = FixedToFP(intval, fracbits, FALSE, FPCR, rounding);
V[d] = fltval;
```
SCVT F (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>sf</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>ftype</th>
<th>1</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>
<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>
</tbody>
</table>
```

rmode opcode
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**

```c
CheckFPAdvSIMEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, 0, FALSE, FPCR, rounding);
V[d] = fltval;
```

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \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. In Armv8.2 and Armv8.3, this is an \texttt{OPTIONAL} instruction. From Armv8.4 it is mandatory for all implementations to support it. \texttt{ID\_AA64ISAR0\_EL1}.DP indicates whether this instruction is supported.

**Vector (Armv8.2)**

\[
\begin{array}{ccccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{size} & L & M & Rm & 1 & 1 & 1 & 0 & H & 0 & Rn & Rd & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & U
\end{array}
\]

**Vector**

\[
\text{SDOT} \langle Vd \rangle, \langle Ta \rangle, \langle Vn \rangle, \langle Tb \rangle, \langle Vm \rangle.4B[\langle index \rangle]
\]

\[
\text{if !HaveDOTPExt()} \text{ then UNDEFINED;}
\text{if size != '10' then UNDEFINED;}
\text{boolean signed = (U == '0');}
\]

\[
\text{integer d = UInt(Rd);} \\
\text{integer n = UInt(Rn);} \\
\text{integer m = UInt(M:Rm);} \\
\text{integer index = UInt(H:L);} \\
\text{integer esize = 8 << UInt(size);} \\
\text{integer datasize = if Q == '1' then 128 else 64;} \\
\text{integer elements = datasize DIV esize;}
\]

**Assembler Symbols**

\[
\text{<Vd> \quad \text{Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.}}
\]

\[
\text{<Ta> \quad \text{Is an arrangement specifier, encoded in "Q":}}
\]

\[
\begin{array}{cc}
Q & \text{<Ta>} \\
0 & 25 \\
1 & 45 \\
\end{array}
\]

\[
\text{<Vn> \quad \text{Is the name of the first SIMD&FP source register, encoded in the "Rn" field.}}
\]

\[
\text{<Tb> \quad \text{Is an arrangement specifier, encoded in "Q":}}
\]

\[
\begin{array}{cc}
Q & \text{<Tb>} \\
0 & 8B \\
1 & 16B \\
\end{array}
\]

\[
\text{<Vm> \quad \text{Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.}}
\]

\[
\text{<index> \quad \text{Is the element index, encoded in the "H:L" fields.}}
\]
Operation

```c
CheckFPAdvSIMDEnabled64();
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09 rc2_1, sve v2019-09 rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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_AAAOISAR0_EL1`.DP indicates whether this instruction is supported.

### Vector (Armv8.2)

![Vector encoded](image)

**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).

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  |

Rm 0 0 0 0 0 0
Rn Rm Rd

Advanced SIMD

SHA1C \(<Qd>\), \(<Sn>\), \(<Vm>\).4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;

Assembler Symbols

\(\langle Qd \rangle\) Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
\(\langle Sn \rangle\) Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
\(\langle Vm \rangle\) Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

\(\text{AArch64.CheckFPAdvSIMDEnabled}()\);

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| Rd | Rn |

Advanced SIMD

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).

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

**Advanced SIMD**

SHA1M \(<Qd>, <Sn>, <Vm>.45\)

integer \(d = \text{UInt}(Rd);\)
integer \(n = \text{UInt}(Rn);\)
integer \(m = \text{UInt}(Rm);\)
if \(! \text{HaveSHA1Ext}()\) 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**

\[
\text{AArch64.CheckFPAdvSIMDEnabled}();
\]

\[
\begin{align*}
\text{bits}(128) \ X &= V[d]; \\
\text{bits}(32) \ Y &= V[n]; & \text{Note: 32 not 128 bits wide} \\
\text{bits}(128) \ W &= V[m]; \\
\text{bits}(32) \ t; \\
\text{for } \ e = 0 \text{ to 3} & \\
& \quad \text{t} = \text{SHA}\text{majority}(X<63:32>, X<95:64>, X<127:96>); \\
& \quad Y = Y + \text{ROL}(X<31:0>, 5) + t + \text{Elem}[W, e, 32]; \\
& \quad X<63:32> = \text{ROL}(X<63:32>, 30); \\
& \quad <Y, X> = \text{ROL}(Y:X, 32); \\
& \quad V[d] = X;
\end{align*}
\]

**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.
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  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Advanced SIMD

SHA1P <Qd>, <Sn>, <Vm>.4S

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

SHA1 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

Advanced SIMD

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  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | Rn |    |    |    |    |    |
| 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | Rd |    |    |    |    |    |

Advanced SIMD

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.
SHA256H2

SHA256 hash update (part 2).

Advanced SIMD

SHA256H2 $<\text{qd}>$, $<\text{qn}>$, $<\text{vm}>$.4S

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;

Assembler Symbols

$<\text{qd}>$ Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
$<\text{qn}>$ Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
$<\text{vm}>$ 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).

Advanced SIMD

SHA256H <Qd>, <Qn>, <Vm>.45

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  |

Advanced SIMD

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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SHA256SU1

SHA256 schedule update 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>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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Advanced SIMD

SHA256SU1 $<Vd>.4S$, $<Vn>.4S$, $<Vm>.4S$

integer $d = \text{UInt}(Rd);$
integer $n = \text{UInt}(Rn);$
integer $m = \text{UInt}(Rm);$
if !\text{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

$\text{AArch64.CheckFPAdvSIMDEnabled}();$

bits(128) operand1 = $V[d];$
bits(128) operand2 = $V[n];$
bits(128) operand3 = $V[m];$
bits(128) result;
bits(128) $T0 = \text{operand3}\langle31:0\rangle:\text{operand2}\langle127:32\rangle;$
bits(64) $T1;$
bits(32) elt;

$T1 = \text{operand3}\langle127:64\rangle;$
for $e = 0$ to 1
    elt = $\text{Elem}[T1, e, 32];$
    elt = $\text{ROR}(elt, 17) \text{ EOR } \text{ROR}(elt, 19) \text{ EOR } \text{LSR}(elt, 10);$ 
    elt = elt + $\text{Elem}[\text{operand1}, e, 32] + \text{Elem}[T0, e, 32];$
$\text{Elem}[\text{result}, e, 32] = elt;$

$T1 = \text{result}\langle63:0\rangle;$
for $e = 2$ to 3
    elt = $\text{Elem}[T1, e-2, 32];$
    elt = $\text{ROR}(elt, 17) \text{ EOR } \text{ROR}(elt, 19) \text{ EOR } \text{LSR}(elt, 10);$ 
    elt = elt + $\text{Elem}[\text{operand1}, e, 32] + \text{Elem}[T0, e, 32];$
$\text{Elem}[\text{result}, e, 32] = elt;$

$V[d] = \text{result};$

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.
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 ARMv8.2-SHA 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>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>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>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>

Advanced SIMD

SHA512H2 <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) NSigma0;
bits(128) X = V[n];
bits(128) Y = V[m];
bits(128) W = V[d];

NSigma0 = ROR(Y<63:0>, 28) EOR ROR(Y<63:0>, 34) EOR ROR(Y<63:0>, 39);
Vtmp<127:64> = (X<63:0> AND Y<127:64>) EOR (X<63:0> AND Y<63:0>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + W<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = (Vtmp<127:64> AND Y<63:0>) EOR (Vtmp<127:64> AND Y<127:64>) EOR (Y<127:64> AND Y<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + 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.
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 ARMv8.2-SHA 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 | Rd 1 0 0 0 0 0 | Rn | Rd
```

Advanced SIMD

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;
bites(64) MSigma1;
bites(64) tmp;
bites(128) X = V[n];
bites(128) Y = V[m];
bites(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 ARMv8.2-SHA 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  | 1  | 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  |
```

Rn Rd

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

```
AArch64.CheckFPAdvSIMDEnabled();
bits(64) sig0;
bits(128) Vtmp;
bits(128) X = V[n];
bits(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 ARMv8.2-SHA 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  |    | Rm |    | 1  | 0  | 0  | 0  | 1  | 0  |    |    |    |    |    |    |    |    |    |    |    |

Advanced SIMD

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.
SHADD

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.

31 30 29 28 27 26 25 24 23 22 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 | 0 | 1 | Rn | Rd |

Three registers of the same type

SHADD <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.
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

| 31 30 29 28 27 26 25 24 23 22 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 1 0 1 0 | Rn | Rd |

Scalar

SHL <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

| 31 30 29 28 27 26 25 24 23 22 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 | != 0000 | immh | 0 1 0 1 0 | Rn | Rd |

Vector

SHL <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

...
<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>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>

<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**

```c
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.

\[
\begin{array}{cccccccccccccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 0 & \text{size} & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & \text{Rn} & \text{Rd} \\
\end{array}
\]

Vector

SHLL(2) `<Vd>..<Ta>, <Vn>..<Tb>, #<shift>

integer \(d = \text{UInt}(Rd)\);
integer \(n = \text{UInt}(Rn)\);

if \(\text{size} == '11'\) then UNDEFINED;
integer \(\text{esize} = 8 \ll \text{UInt}(\text{size})\);
integer \(\text{datasize} = 64\);
integer \(\text{part} = \text{UInt}(Q)\);
integer \(\text{elements} = \text{datasize} \div \text{esize}\);

integer \(\text{shift} = \text{esize}\);
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":

\[
\begin{array}{c|c}
\text{Q} & 2 \\
0 & \text{[absent]} \\
1 & \text{[present]} \\
\end{array}
\]

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<Ta>` Is an arrangement specifier, encoded in "size":

\[
\begin{array}{c|c}
\text{size} & \text{<Ta>} \\
00 & \text{8H} \\
01 & \text{4S} \\
10 & \text{2D} \\
11 & \text{RESERVED} \\
\end{array}
\]

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

`<Tb>` Is an arrangement specifier, encoded in "size:Q":

\[
\begin{array}{c|c|c}
\text{size} & \text{Q} & \text{<Tb>} \\
00 & 0 & \text{8B} \\
00 & 1 & \text{16B} \\
01 & 0 & \text{4H} \\
01 & 1 & \text{8H} \\
10 & 0 & \text{2S} \\
10 & 1 & \text{4S} \\
11 & x & \text{RESERVED} \\
\end{array}
\]

`<shift>` Is the left shift amount, which must be equal to the source element width in bits, encoded in "size":

\[
\begin{array}{c|c}
\text{size} & \text{<shift>} \\
00 & 8 \\
01 & 16 \\
10 & 32 \\
11 & \text{RESERVED} \\
\end{array}
\]
Operation

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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | Q  | 0  | 1  | 1  | 1  | 1  | O  | != | 0000 | immh | 1  | 0  | 0  | 0  | 0  | 1  | Rn  | Rd  |
|иммh|оп |
```

**Vector**

```
Vector(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":

<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>
<th>SEE Advanced SIMD modified immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>8B</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>0</td>
<td>4H</td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>1</td>
<td>8H</td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>0</td>
<td>2S</td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>4S</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>x</td>
<td></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>
<th>SEE Advanced SIMD modified immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>8H</td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>4S</td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>2D</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td></td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
```
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**

```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.
SHSUB

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 | 0 | size | 1 | Rm | 0 | 0 | 1 | 0 | 0 | 1 | Rn | 0 | 0 | Rd |

Three registers of the same type

SHSUB <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 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.
SLI

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.

```
\[\begin{array}{cccccccc}
63 & 56 & 55 & 54 & 53 & 52 & 51 & 50 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0
\end{array}\]
```

```
\[\begin{array}{cccccccc}
63 & 56 & 55 & 54 & 53 & 52 & 51 & 50 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0
\end{array}\]
```

Depending on the settings in the `CPACR_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 1 1 0 != 0000
```

Scalar

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**

```
0 Q 1 0 1 1 1 1 0 != 0000
```

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

Scalar
Vector

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>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>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 ARMv8.2-SM 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 1 0 0 0 0 1 1 0 0 0 0 1 1 |

Rm | Rn | Rd |
```

Advanced SIMD

SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```python
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
in integer n = UInt(Rn);
in 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);
  else
    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);
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 ARMv8.2-SM 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  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  |
| Rd |

Advanced SIMD

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 ARMv8.2-SM 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  | 0  | Ra  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| 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 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 ARMv8.2-SM 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>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>
</tr>
</tbody>
</table>

**SM3TT1A**

\[
\begin{align*}
\text{SM3TT1A} & \quad <Vd>.4S, \quad <Vn>.4S, \quad <Vm>.S[<imm2>] \\
\text{if !HaveSM3Ext()} & \text{then UNDEFINED;} \\
n & = \text{UInt}(Rn); \\
m & = \text{UInt}(Rm); \\
i & = \text{UInt}(imm2); \\
\end{align*}
\]

**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**

\[
\begin{align*}
\text{AArch64.CheckFPAdvSIMDEnabled}(); \\
\text{bits}(128) \quad WjPrime & = \text{Elem}[Vm, i, 32]; \\
\text{SS2} & = Vn<127:96> \ \text{EOR} \ \text{ROL}(Vd<127:96>, 12); \\
\text{TT1} & = Vd<63:32> \ \text{EOR} \ (Vd<127:96> \ \text{EOR} \ \text{Vd}<95:64>); \\
\text{TT1} & = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>; \\
\text{result}<31:0> & = Vd<63:32>; \\
\text{result}<63:32> & = \text{ROL}(Vd<95:64>, 9); \\
\text{result}<95:64> & = Vd<127:96>; \\
\text{result}<127:96> & = TT1; \\
V[d] & = \text{result};
\end{align*}
\]
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.
SM3TT1B

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 ARMv8.2-SM 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 1 | Rn | Rd |

Advanced SIMD

SM3TT1B <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.CheckFPAdySIMDEnabled();

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<127:96> AND Vd<63:32>) OR (Vd<127:96> AND Vd<95:64>) OR (Vd<63:32> AND 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 values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 ARMv8.2-SM is implemented.

Advanced SIMD
(Armv8.2)

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0</td>
<td>0 1 1</td>
<td>Vd</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1 0 0</td>
<td>Rn</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rd</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.CheckFPAvSIMDEnabled();

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 ARMv8.2-SM is implemented.

Advanced SIMD
(Armv8.2)

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 ARMv8.2-SM 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  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |

Advanced SIMD

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;
roundresult = V[d];
for index = 0 to 3
    roundkey = 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<31:0> = roundresult<63:32>;
roundresult<95:64> = roundresult<95:64>;
roundresult<127:96> = 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 ARMv8.2-SM 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  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

**Advanced SIMD**

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]);
    end
    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.

Three registers of the same type

SMAX <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.
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.

Three registers of the same type

SAXMП <Vd>..<T>, <Vs>..<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>

<Vs> 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 = Vs[n];
bits(datasize) operand2 = Vs[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>;
VS[d] = result;
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 **CPACR_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 1 1 0 1 0 0 0 0 1 0 1 0 1 0
U  op

Rn Rd
```

### Advanced SIMD

**SMAXV <V>, <d>, <Vn>,<T>**

integer \( d = \text{UInt}(\text{Rd}) \);
integer \( n = \text{UInt}(\text{Rn}) \);

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');

### Assembler Symbols

- **<V>** Is the destination width specifier, encoded in “size”:

```
size <V>
00 B
01 H
10 S
```

- **<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”:

```
size Q <T>
00 0 8B
00 1 16B
01 0 4H
01 1 8H
10 0 RESERVED
10 1 4S
11 x RESERVED
```

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
integer maxmin;
integer element;

maxmin = Int(\text{Elem}[\text{operand}, 0, \text{esize}], \text{unsigned});
for e = 1 to elements-1
    element = Int(\text{Elem}[\text{operand}, e, \text{esize}], \text{unsigned});
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);
V[d] = maxmin^esize-1:0>;
```

SMAXV
Operational information

If PSTATE.DIT is 1:

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

- The response of this instruction to 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.

```
0 0 0 0 1 1 1 0 | size:1 | Rm:0 1 1 0 | Rd:1 1 0 1 | o1
```

Three registers of the same type

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` ÷ `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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Rm</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>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>
</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>
```

Three registers of the same type

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.
SMINV

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 CPACR_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 1 1 0 0 0 1 1 0 1 0  Rd
0 0 0 0 1 1 1 0  U  op
```

Advanced SIMD

SMINV <V><d>, <Vn>.<T>

```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');
```

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>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

```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>;
```
Operational information

If PSTATE.DIT is 1:

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

- The response of this instruction to 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 (by element)

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.

Vector

SMLAL{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>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.
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.

Three registers, not all the same type

SMLAL{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

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.
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.

```
integer idxdsise = 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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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":

```
<Ta>

<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;

```c
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
| Q | 0 | 0 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 1 | 0 | 0 | 0 | Rn | Rd |

Three registers, not all the same type

SMLSL(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

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

\[
\text{CheckFPAdvSIMDEnabled64}();
\]

\[
\begin{align*}
\text{bits}(&\text{datasize}) \text{ operand1} = \text{Vpart}[n, \text{part}]; \\
\text{bits}(&\text{datasize}) \text{ operand2} = \text{Vpart}[m, \text{part}]; \\
\text{bits}(2*\text{datasize}) \text{ operand3} = \text{V}[d]; \\
\text{bits}(2*\text{datasize}) &\text{ result} \\
\text{integer} &\text{ element1} \\
\text{integer} &\text{ element2} \\
\text{bits}(2*\text{esize}) &\text{ product} \\
\text{bits}(2*\text{esize}) &\text{ accum} \\
\end{align*}
\]

\[
\text{for} \ e = 0 \ \text{to} \ \text{elements-1} \\
\quad \text{element1} = \text{Int(Elem}[\text{operand1}, e, \text{esize}], \text{unsigned}); \\
\quad \text{element2} = \text{Int(Elem}[\text{operand2}, e, \text{esize}], \text{unsigned}); \\
\quad \text{product} = (\text{element1}*\text{element2})<2*\text{esize}-1:0>; \\
\quad \text{if} \ \text{sub_op} \ \text{then} \\
\quad \quad \text{accum} = \text{Elem}[\text{operand3}, e, 2*\text{esize}] - \text{product}; \\
\quad \text{else} \\
\quad \quad \text{accum} = \text{Elem}[\text{operand3}, e, 2*\text{esize}] + \text{product}; \\
\quad \text{Elem}[\text{result}, e, 2*\text{esize}] = \text{accum}; \\
\]

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

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.
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_AA64ISBAR0_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  | 1  | 1  | 1  | 0  | 1  | 0  | 0  |    |   |   |    |   |   |    |   |   |    |   |   |    |   |   |    |   |   |    |   |   |    |   |   |    |   |   |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

U

B

Vector

SMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

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

CheckFPAdvSIMDEnabled64();

bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];

V[d] = MatMulAdd(addend, operand1, operand2, FALSE, FALSE);
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 \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}
0 | Q | Q0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | imm5 | 0 | 0 | 1 | 0 | 1 | 1 | Rn | Rd
\end{verbatim}

\textbf{32-bit (Q == 0)}

\textbf{SMOV} \texttt{<Wd>}, <Vn>.<Ts>[<index>]

\textbf{64-reg, SMOV-64-reg (Q == 1)}

\textbf{SMOV} \texttt{<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 index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;

\textbf{Assembler Symbols}

\texttt{<Wd>} Is the 32-bit name of the general-purpose destination register, encoded in the “Rd” field.
\texttt{<Xd>} Is the 64-bit name of the general-purpose destination register, encoded in the “Rd” field.
\texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the “Rn” field.
\texttt{<Ts>} For the 32-bit variant: is an element size specifier, encoded in “imm5”:

\begin{verbatim}
xxx00    RESERVED
xxx1     B
xxx10    H
xxxx10   S
\end{verbatim}

For the 64-reg, SMOV-64-reg variant: is an element size specifier, encoded in “imm5”:

\begin{verbatim}
xx000    RESERVED
xxx1     B
xxx10    H
xx100    S
\end{verbatim}

\texttt{<index>} For the 32-bit variant: is the element index encoded in “imm5”:

\begin{verbatim}
xxx00    RESERVED
xxx1     imm5<4:1>
xxx10    imm5<4:2>
\end{verbatim}

For the 64-reg, SMOV-64-reg variant: is the element index encoded in “imm5”:
### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n];
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.
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.

---

**Vector**

\[
\text{SMULL2} \ (\text{by element})
\]

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>

\(<\text{Vd}>\) Is the name of the SIMD&FP destination register, encoded in the "Rd" field.  
\(<\text{Ta}>\) Is an arrangement specifier, encoded in “size”:

<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>

\(<\text{Vn}>\) Is the name of the first 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>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) 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.

Three registers, not all the same type

SMULL{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**

```plaintext
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

<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>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 1 0</td>
<td>1 0 0 0 0 0</td>
<td>0 0 1 1 1 1 0</td>
<td>1 0 0 0 0 0 0 1 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

#### Scalar

**SQABS** $<V>d$, $<V>n$

```plaintext
tagger
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>Rd</th>
<th>Rn</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 1 1 0</td>
<td>1 0 0 0 0 0</td>
<td>0 0 1 1 1 1 0</td>
<td>1 0 0 0 0 0 0 1 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

#### Vector

**SQABS** $<Vd>.<T>$, $<Vn>.<T>$

```plaintext
tagger
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>value</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
CheckFPAvSIMDEnabled64();
bias(datasize) operand = V[n];
bias(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

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

Scalar 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

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

Vector 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

<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>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>10B</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];
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 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  | 0   | 0   | 1   | 1   | H   | 0   | Rn  | Rd  | o2  |

### Scalar

SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

integer idxdsz = 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 datasz = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '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   | O   | 0   | 1   | 1   | 1   | size | L   | M   | Rm  | 0   | 0   | 1   | 1   | H   | 0   | Rn  | Rd  | o2  |

SQDMLAL, SQDMLAL2 (by element)
Vector

SQDMLAL[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 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.
Is the name of the second SIMD&FP source register, encoded in "size:M:Rm":

<table>
<thead>
<tr>
<th>size</th>
<th>M:Rm</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":

<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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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**

| 31 30 29 28 27 26 25 24 23 22 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 0 1 0 0 | Rn | Rd |

01

**Scalar**

$SQDMLAL \langle Va \rangle_1 \langle d \rangle, \langle Vb \rangle_0 \langle n \rangle, \langle Vb \rangle_0 \langle m \rangle$

```plaintext
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**

| 31 30 29 28 27 26 25 24 23 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 1 0 0 | Rn | Rd |

01

**Vector**

$SQDMLAL\{2\} \langle Vd \rangle_1 \langle Ta \rangle, \langle Vn \rangle_0 \langle Tb \rangle, \langle Vm \rangle_0 \langle Tb \rangle$

```plaintext
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 part = UInt(Q);
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>

<\(V_d\)>  Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<\(T_a\)>  Is an arrangement specifier, encoded in "\(\text{size}\)"

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;(T_a)&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>

<\(V_n\)>  Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<\(T_b\)>  Is an arrangement specifier, encoded in "\(\text{size}:Q\)"

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;(T_b)&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>

<\(V_m\)>  Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<\(V_a\)>  Is the destination width specifier, encoded in "\(\text{size}\)"

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;(V_a)&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.

<\(V_b\)>  Is the source width specifier, encoded in "\(\text{size}\)"

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;(V_b)&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

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

| 31 30 29 28 27 26 25 24 23 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       |      1       |      size     |      L       |      M       |      Rm      |      0       |      1       |      1       |      1       |      H       |      0       |      Rn       |      Rd       |      0       |      1       |      1       |      1       |      H       |      0       |

Scalar

SQDMLSL <Va><d>, <Vb><n>, <Vm><Ts>[<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 = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '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       |      size     |      L       |      M       |      Rm      |      0       |      1       |      1       |      1       |      H       |      0       |      Rn       |      Rd       |      0       |      1       |      1       |      1       |      H       |      0       |      Rn       |      Rd       |      0       |      1       |      1       |      1       |      H       |      0       |      Rn       |      Rd       |
Vector

SQDMLSL[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 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></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>
<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;
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 1 1 0 size 1 Rm</td>
</tr>
<tr>
<td>01</td>
</tr>
</tbody>
</table>

**Scalar**

$\text{SQDMLSL } <\text{Va}><d>, <\text{Vb}><n>, <\text{Vb}><m>$

```plaintext
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>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 1 0 size 1 Rm</td>
</tr>
<tr>
<td>01</td>
</tr>
</tbody>
</table>

**Vector**

$\text{SQDMLSL2 } <\text{Vd}>, <\text{Ta}>, <\text{Vn}>, <\text{Tb}>, <\text{Vm}>, <\text{Tb}>$

```plaintext
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 part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```
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

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;
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 SQRDMULH.

Depending on the settings in the CPACR_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  | L  | M  | Rm | 1  | 1  | 0  | 0  | H  | 0  | 0  | Rn | Rd |

op

Scalar

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  | 0  | 0  | 1  | 1  | 1  | 1  | L  | M  | Rm | 1  | 1  | 0  | 0  | H  | 0  | 0  | Rn | Rd |

op
Vector

\[ \text{SQDMULH } \langle Vd \rangle.<T>, \langle Vn \rangle.<T>, \langle Vm \rangle.<Ts>[<index>] \]

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;

\text{case size of}
\begin{align*}
\quad & \text{when '01' index = Uint}(H:L:M); Rmhi = '0'; \\
\quad & \text{when '10' index = Uint}(H:L); Rmhi = M; \\
\quad & \text{otherwise UNDEFINED;}
\end{align*}

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

\text{<V>} is a width specifier, encoded in “size”:

\begin{center}
\begin{tabular}{ccc}
\text{size} & \text{<V>} \\
00 & RESERVED \\
01 & H \\
10 & S \\
11 & RESERVED \\
\end{tabular}
\end{center}

\text{<d>} is the number of the SIMD&FP destination register, encoded in the "Rd" field.

\text{<n>} is the number of the first SIMD&FP source register, encoded in the "Rn" field.

\text{<Vd>} is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\text{<T>} is an arrangement specifier, encoded in “size:Q”:

\begin{center}
\begin{tabular}{ccc}
\text{size} & \text{Q} & \text{<T>} \\
00 & x & RESERVED \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & RESERVED \\
\end{tabular}
\end{center}

\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 “size:M:Rm”:

\begin{center}
\begin{tabular}{ccc}
\text{size} & \text{<Vm>} \\
00 & RESERVED \\
01 & 0:Rm \\
10 & M:Rm \\
11 & RESERVED \\
\end{tabular}
\end{center}

Restricted to V0-V15 when element size <Ts> is H.

\text{<Ts>} is an element size specifier, encoded in “size”:

\begin{center}
\begin{tabular}{ccc}
\text{size} & \text{<Ts>} \\
00 & RESERVED \\
01 & H \\
10 & S \\
11 & RESERVED \\
\end{tabular}
\end{center}

\text{<index>} is the element index, encoded in “size:L:H:M”:
Operation

CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n];
bits(idxsizsize) 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;
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 SQRDMLH.
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

\[
\begin{array}{ccccccccccccccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & \text{size} & 1 & \text{Rm} & 1 & 0 & 1 & 1 & 0 & 1 & \text{Rn} & \text{Rd} & U
\end{array}
\]

Scalar

\[
\text{SQDMULH} <V><d>, <V><n>, <V><m>
\]

integer \( d = \text{UInt}(Rd); \)
integer \( n = \text{UInt}(Rn); \)
integer \( m = \text{UInt}(Rm); \)
if size == '11' || size == '00' then UNDEFINED;
integer \( \text{esize} = 8 << \text{UInt}(size); \)
integer \( \text{datasize} = \text{esize}; \)
integer \( \text{elements} = 1; \)
boolean \( \text{rounding} = (U == '1'); \)

Vector

\[
\begin{array}{ccccccccccccccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & \text{size} & 1 & \text{Rm} & 1 & 0 & 1 & 1 & 0 & 1 & \text{Rn} & \text{Rd} & U
\end{array}
\]

Vector

\[
\text{SQDMULH} <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
\]

integer \( d = \text{UInt}(Rd); \)
integer \( n = \text{UInt}(Rn); \)
integer \( m = \text{UInt}(Rm); \)
if size == '11' || size == '00' then UNDEFINED;
integer \( \text{esize} = 8 << \text{UInt}(size); \)
integer \( \text{datasize} = \text{if } Q == '1' \text{ then } 128 \text{ else } 64; \)
integer \( \text{elements} = \text{datasize} \text{ DIV esize}; \)
boolean \( \text{rounding} = (U == '1'); \)

Assembler Symbols

<\text{V}> 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>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{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>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 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 = 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;
```

Copyright © 2010-2019 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

```
size L M Rm 1 0 1 1 H 0 Rn Rd
```

Scalar

```plaintext
SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]
```

integer idxdsig = 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

```
size L M Rm 1 0 1 1 H 0 Rn Rd
```

SQDMULL, SQDMULL2 (by element)
Vector

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”: 

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>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) 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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 1  | 0  | 0  | Rn  | Rd  |
```

Scalar

```
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 = 8;

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 | Rm  | 1 | 1 | 0 | 1 | 0 | 0 | Rn  | Rd  |
```

Vector

```
SQDMULL[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 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":
```
<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vb> Is an arrangement specifier, encoded in "size:Q":

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
</tr>
</thead>
</table>
| 00   | x   | RESERVED
| 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.

<Va> Is the destination width specifier, encoded in "size":

<table>
<thead>
<tr>
<th>size</th>
</tr>
</thead>
</table>
| 00   | RESERVED
| 01   | S
| 10   | D
| 11   | RESERVED

<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>
</tr>
</thead>
</table>
| 00   | RESERVED
| 01   | H
| 10   | S
| 11   | RESERVED

<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();
bv<datasize>(operand1) = Vpart[n, part];
bv<datasize>(operand2) = Vpart[m, part];
bv<2*datasize>(result);
integer element1;
integer element2;
bv<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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQNEG

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

31 30 29 28 27 26 25 24 23 22 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 0 1 1 1 1 0 | Rd  |

U

Scalar

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

31 30 29 28 27 26 25 24 23 22 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 0 0 0 0 | 0 0 1 1 1 1 0 | Rd  |

U

Vector

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>
</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>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bites(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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)

```
| 31 30 29 28 27 26 25 24 23 22 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 | size | L | M | Rm | 1 | 1 | 0 | 1 | H | 0 | Rn | Rd |
```

Scalar

SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;

```hascal
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 | Q | 1 | 0 | 1 | 1 | 1 | 1 | size | L | M | Rm | 1 | 1 | 0 | 1 | H | 0 | Rn | Rd |
```
Vector

SQRDMLAH <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

\[ \text{CheckFPAdvSIMDEnabled64}(); \]

\[
\begin{align*}
\text{bits(datasize)} & \text{ operand1} = V[n]; \\
\text{bits(idxdsize)} & \text{ operand2} = V[m]; \\
\text{bits(datasize)} & \text{ operand3} = V[d]; \\
\text{bits(datasize)} & \text{ result}; \\
\text{integer} \quad & \text{rounding\_const} = \text{if\ rounding\ then\ } 1 \ll (\text{esize} - 1) \text{ else 0; } \\
\text{integer} \quad & \text{element1; } \\
\text{integer} \quad & \text{element2; } \\
\text{integer} \quad & \text{element3; } \\
\text{integer product; } \\
\text{boolean sat; } \\
\end{align*}
\]

\[
\begin{align*}
\text{element2} & = \text{SInt}(\text{Elem}[\text{operand2, index, esize}]); \\
\text{for } e = 0 & \text{ to elements}\_1 \text{ do } \\
\text{element1} & = \text{SInt}(\text{Elem}[\text{operand1, e, esize}]); \\
\text{element3} & = \text{SInt}(\text{Elem}[\text{operand3, e, esize}]); \\
\text{if sub\_op then } & \\
\text{accum} & = ((\text{element3} \ll \text{esize}) - 2 * (\text{element1} * \text{element2}) + \text{rounding\_const}); \\
\text{else } & \\
\text{accum} & = ((\text{element3} \ll \text{esize}) + 2 * (\text{element1} * \text{element2}) + \text{rounding\_const}); \\
(\text{Elem}[\text{result, e, esize}], \text{sat}) & = \text{SignedSatQ}(\text{accum} \gg \text{esize}, \text{esize}); \\
\text{if sat then } & \\
\text{FPSR.QC} & = '1'; \\
\end{align*}
\]

\[ V[d] = \text{result}; \]
SQRDMALAH (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)

```
+--------------------------------+}
| 31 30 29 28 27 26 25 24 23 22 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 | 0 | Rm | 1 0 0 0 | 0 1 | Rn | Rd |
|--------------------------------+}
```

**Scalar**

SQRDMLAH <V><d>, <V><n>, <V><m>

if !HaveQRDMLAHExte() then UNDEFINED;

```java
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)

```
+--------------------------------+}
| 31 30 29 28 27 26 25 24 23 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 | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
|--------------------------------+}
```

**Vector**

SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

if !HaveQRDMLAHExte() then UNDEFINED;

```java
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":

SQRDMLAH (vector)
<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>&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>

<table>
<thead>
<tr>
<th>size &lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 x</td>
</tr>
<tr>
<td>01 0</td>
</tr>
<tr>
<td>01 1</td>
</tr>
<tr>
<td>10 0</td>
</tr>
<tr>
<td>10 1</td>
</tr>
<tr>
<td>11 x</td>
</tr>
</tbody>
</table>

<br>

<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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | L  | M  | Rm | 1  | 1  | 1  | 1  | H  | 0  | Rn | Rd |

Scalar

SQRDM LSH <V><d>, <V><n>, <Vm><Ts><index>

if !HaveQRDMLAHExt() then UNDEFINED;

integer idedxsize = 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  | Q  | 1  | 0  | 1  | 1  | 1  | 1  | L  | M  | Rm | 1  | 1  | 1  | 1  | H  | 0  | Rn | Rd |

SQRDMLSH (by element)  Page 1089
Vector

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
CheckFPAdvSIMEnabled64();
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SQRDMLSH (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)

| 31 30 29 28 27 26 25 24 23 22 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 | Rm | 1 0 0 0 1 1 | Rn | Rd |

**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 Q 1 0 1 1 1 0 | size | Rm | 1 0 0 0 1 1 | Rn | Rd |

**Assembler Symbols**

<V> Is a width specifier, encoded in “size”:
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>00</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>00</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<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 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) 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 SQRDMULH.

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**

31 30 29 28 27 26 25 24 23 22 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 size | L | M | Rm | 1 1 0 1 H | 0 Rn | Rd | op

**Vector**

0 0 0 0 1 1 1 1 size | L | M | Rm | 1 1 0 1 H | 0 Rn | Rd | op

Scalar

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

SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

integer idxdsz = 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**

```c
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

| 31 30 29 28 27 26 25 24 23 22 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 | 1 | 0 | 1 | 0 | Rn | 1 | Rd | U |

Scalar

SQRDMULH <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 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 1 | 0 | 1 | 0 | Rn | 1 | Rd | U |

Vector

SQRDMULH <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.
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>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 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 = 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \texttt{SQSRL}.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \texttt{FPSR.QC} is set.

Depending on the settings in the \texttt{CPACR	extunderscore EL1}, \texttt{CPTR	extunderscore EL2}, and \texttt{CPTR	extunderscore 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{Scalar} and \texttt{Vector}

**Scalar**

\begin{tabular}{|c|c|c|c|}
\hline
31 & 30 & 29 & 28  \\
0 & 1 & 1 & 1  \\
\hline
size & 1 & Rm & \hline
0 & 1 & 0 & 1  \\
\hline
Rn & Rd & \hline
0 & 1 & 1 & 1  \\
\hline
U & R & S & \hline
\end{tabular}

**Scalar**

\texttt{SQRSHL <V>d>, <V<n>, <V><m>}

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
integer esize = 8 << \texttt{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**

\begin{tabular}{|c|c|c|c|}
\hline
31 & 30 & 29 & 28  \\
0 & 1 & 1 & 1  \\
\hline
size & 1 & Rm & \hline
0 & 1 & 0 & 1  \\
\hline
Rn & Rd & \hline
0 & 1 & 1 & 1  \\
\hline
U & R & S & \hline
\end{tabular}

**Vector**

\texttt{SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
if size:Q == '110' 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 rounding = (R == '1');
boolean saturating = (S == '1');

**Assembler Symbols**

\texttt{<V>} Is a width specifier, encoded in "size":

SQRSHL
<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>&lt;T&gt;</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>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 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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**, **CPTTR_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
Scalar

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**

```plaintext
Vector

SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

integer d = UInt(Rd);
integer n = UInt(Rn);
if immm == '0000' then SEE(asimdimm);
if immm<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
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 “im mh: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 “im mh”:

<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 “im mh”:

<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 “im mh”:

<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 “im mh:im mb”:

<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(im mh:im mb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(im mh:im mb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(im mh:im mb))</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 “im mh:im mb”:

<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(im mh:im mb))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(im mh:im mb))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(im mh:im mb))</td>
</tr>
<tr>
<td>1xxx</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
b_bits(datasize*2) operand = V[n];
b_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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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**

\[
\begin{array}{c|cccccc|c|c}
0 & 1 & 1 & 1 & 1 & 1 & 0 & \text{!} & 0 & 0 & 0 & 0 & 1 & 1 & \text{Rn} & \text{Rd}
\end{array}
\]

**Scalar**

\[
\text{SQRSHRUN} <Vb><d>, <Va><n>, #<shift>
\]

\[
\begin{align*}
\text{integer } d &= \text{UInt}(\text{Rd}); \\
\text{integer } n &= \text{UInt}(\text{Rn}); \\
\text{if } \text{immh} == '0000' \text{ then UNDEFINED;} \\
\text{if } \text{immh}<3> == '1' \text{ then UNDEFINED;} \\
\text{integer esize} &= 8 \ll \text{HighestSetBit}(\text{immh}); \\
\text{integer datasize} &= \text{esize}; \\
\text{integer elements} &= 1; \\
\text{integer part} &= 0;
\end{align*}
\]

\[
\text{integer shift} = (2 \times \text{esize}) - \text{UInt}(\text{immh:immb}); \\
\text{boolean round} = (\text{op} == '1');
\]

**Vector**

\[
\begin{array}{c|cccccc|c|c}
0 & 0 & 1 & 0 & 1 & 1 & 1 & \text{!} & 0 & 0 & 0 & 1 & 1 & \text{Rn} & \text{Rd}
\end{array}
\]

**Vector**

\[
\text{SQRSHRUN(2)} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>
\]

\[
\begin{align*}
\text{integer } d &= \text{ UInt}(\text{Rd}); \\
\text{integer } n &= \text{UInt}(\text{Rn}); \\
\text{if } \text{immh} == '0000' \text{ then SFE(asimdimm);} \\
\text{if } \text{immh}<3> == '1' \text{ then UNDEFINED;} \\
\text{integer esize} &= 8 \ll \text{HighestSetBit}(\text{immh}); \\
\text{integer datasize} &= 64; \\
\text{integer elements} &= \text{datasize DIV esize}; \\
\text{integer part} &= \text{UInt}(Q); \\
\text{integer shift} &= (2 \times \text{esize}) - \text{UInt}(\text{immh:immb}); \\
\text{boolean round} &= (\text{op} == '1');
\end{align*}
\]
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>Tb</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>lxxx</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>lxxx</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>lxxx</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>lxxx</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:immh”:

<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:immh))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immh))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immh))</td>
</tr>
<tr>
<td>lxxx</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:immh”:

<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:immh))</td>
</tr>
<tr>
<td>001x</td>
<td>(32-UInt(immh:immh))</td>
</tr>
<tr>
<td>01xx</td>
<td>(64-UInt(immh:immh))</td>
</tr>
<tr>
<td>lxxx</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;
**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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | immh | op |
```

**Scalar**

```
SQSHL <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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 |
```

**Vector**

```
SQSHL (immediate)
```
Vector

SQSHL \textlangle Vd \rangle , \textlangle T \rangle , \textlangle Vn \rangle , \textlangle T \rangle , \# \langle \text{shift} \rangle

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

if immh == '0000' then \texttt{SEE\text{(asimdimm)}};
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 \ll \text{HighestSetBit}(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize \div esize;

integer shift = \text{UInt}(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;

\text{case op:U of}
\text{when '00' UNDEFINED;}
\text{when '01' src_unsigned = FALSE; dst_unsigned = TRUE;}
\text{when '10' src_unsigned = FALSE; dst_unsigned = FALSE;}
\text{when '11' src_unsigned = TRUE; dst_unsigned = TRUE;}

\textbf{Assembler Symbols}

\textlangle V \rangle \quad \text{Is a width specifier, encoded in “immh”:}

\begin{tabular}{c|c}
\textbf{immh} & \textlangle V \rangle \\
\hline
0000 & RESERVED \\
0001 & B \\
001x & H \\
01xx & S \\
1xxx & D \\
\end{tabular}

\textlangle d \rangle \quad \text{Is the number of the SIMD&FP destination register, in the "Rd" field.}

\textlangle n \rangle \quad \text{Is the number of the first SIMD&FP source register, encoded in the "Rn" field.}

\textlangle Vd \rangle \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.}

\textlangle T \rangle \quad \text{Is an arrangement specifier, encoded in “immh:Q”:}

\begin{tabular}{c|c}
\textbf{immh} & \langle T \rangle \\
\hline
0000 & x \quad \texttt{SEE\text{ Advanced SIMD modified immediate}} \\
0001 & 0 \quad 88 \\
0001 & 1 \quad 16B \\
001x & 0 \quad 4H \\
001x & 1 \quad 8H \\
01xx & 0 \quad 2S \\
01xx & 1 \quad 4S \\
1xxx & 0 \quad RESERVED \\
1xxx & 1 \quad 2D \\
\end{tabular}

\textlangle Vn \rangle \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.}

\textlangle \text{shift} \rangle \quad \text{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”:}

\begin{tabular}{c|c}
\textbf{immh} & \langle \text{shift} \rangle \\
\hline
0000 & RESERVED \\
0001 & \text{UInt}(immh:immb)-8 \\
001x & \text{UInt}(immh:immb)-16 \\
01xx & \text{UInt}(immh:immb)-32 \\
1xxx & \text{UInt}(immh:immb)-64 \\
\end{tabular}

\text{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”:}
### Operation

```plaintext
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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, 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 1 0 0 1 1 | Rn | Rd
U R S
```

```
Scalar
```

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

```
0 0 0 1 1 1 1 0 | size | Rm 0 1 0 0 1 1 | Rn | Rd
U R S
```

```
Vector
```

SQSHL <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”:

SQSHL (register)
<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;
    else
        (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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 1 1 1 1 0 != 0000</td>
</tr>
</tbody>
</table>

**Scalar**

```plaintext
SQSHLU <V><d>, <V><n>, #<shift>
```

```plaintext
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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>U</th>
<th>immh</th>
<th>op</th>
</tr>
</thead>
</table>

SQSHLU
Vector

\texttt{SVSLHU <Vd>.<T>, <Vn>.<T>, #<shift>}

integer \( d = \text{UInt}(Rd) \);
integer \( n = \text{UInt}(Rn) \);

if \( \text{immh} == '0000' \) then \text{SEE(\text{asimdimm})};
if \( \text{immh}<3>:Q == '10' \) then \text{UNDEFINED};
integer \( \text{esize} = 8 << \text{HighestSetBit}(\text{immh}) \);
integer \( \text{datasize} = \text{if } Q == '1' \) then 128 else 64;
integer \( \text{elements} = \text{datasize} \div \text{esize} \);

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' \text{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>(Q)</th>
<th>(&lt;T&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>x</td>
<td>\text{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.

\(<\text{shift}>\) 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;\text{shift}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0001</td>
<td>(\text{UInt}(\text{immh}:\text{immb})-8)</td>
</tr>
<tr>
<td>001x</td>
<td>(\text{UInt}(\text{immh}:\text{immb})-16)</td>
</tr>
<tr>
<td>01xx</td>
<td>(\text{UInt}(\text{immh}:\text{immb})-32)</td>
</tr>
<tr>
<td>1xxx</td>
<td>(\text{UInt}(\text{immh}:\text{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”:
### Operation

```c
CheckFPAdvSIMDEnabled64();
binary(datasize) operand = V[n];
binary(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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

<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>! = 0000</td>
<td>immh</td>
<td>1</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>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td>immh</td>
<td></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>
<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>

Scalar

SQSHRN <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

<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>1</td>
<td>0</td>
<td>! = 0000</td>
<td>immh</td>
<td>1</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>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td>immh</td>
<td></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>
<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>

Vector

SQSHRN(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');
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();
b bits(datasize*2) operand = V[n];
b 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 **SQRSHRUN**.

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

**Scalar**

**SQSHRUN <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

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

**Vector**

**SQSHRUN[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 elements = datasize DIV esize;

integer shift = (2 * esize) - **UInt**(immh:immb);
boolean round = (op == '1');
Assembler Symbols

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>Tb</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>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();
b既有(data_size*2) operand = V[n];
b既有(data_size) 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

```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 0 | size 1 | Rm 0 0 1 0 1 1 | Rn | Rd
   U
```

Vector

```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 | Q | 0 0 1 1 1 0 | size 1 | Rm 0 0 1 0 1 1 | Rn | Rd
   U
```

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 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | 1  | 0  | size| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | Rd |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | U  |

#### Scalar

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  | Q | 0  | 0  | 1  | 1  | 1  | 0  | size| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | Rd |
|    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | U  |

#### Vector

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":

Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Is an arrangement specifier, encoded in "size:".

Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Is an arrangement specifier, encoded in "size:".

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 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;
```
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

<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 1 0 0 1 0 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

```plaintext
Scalar

$SQXTUN <Vb><d>$, $<Va><n>$

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

<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 1 0 0 1 0 1 0</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

```plaintext
Vector

$SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>$

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>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>O</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>[absent]</td>
</tr>
<tr>
<td>1</td>
<td>[present]</td>
</tr>
</tbody>
</table>

$v<\text{d}>$ Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

$v<\text{b}>$ Is an arrangement specifier, encoded in "size:Q":

$V<\text{b}>$
<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) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';
Vpart[d, part] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | size | 1  | Rm | 0  | 0  | 0  | 1  | 0  | 1  | Rd |
| U  |

### Three registers of the same type

**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 datatize = if Q == '1' then 128 else 64;
integer elements = datatize 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

```c
CheckFPAdvSIMDEnabled64();
bits(datatize) operand1 = V[n];
bits(datatize) operand2 = V[m];
bits(datatize) 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;
```

---

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SRI**

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.

![Diagram showing 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  0  != 0000  immh 0  1  0  0  0  1  Rd
```

**Scalar**

```plaintext
SRI <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 = (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  0  1  1  1  1  1  1  1  0  != 0000  immh 0  1  0  0  0  1  Rd
```

```
```
Vector

SRI $<Vd> . <T>, <Vn> . <T>$, $\# <shift>$

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;

if immh == '0000' then $\text{SEE(asimdimm)}$;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 $\ll$ $\text{HighestSetBit}(\text{immh})$;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize $\div$ esize;

integer shift = (esize * 2) - $\text{UInt}(\text{immh:immb})$;

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>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>$\text{SEE Advanced SIMD modified immediate}$</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 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-$\text{UInt}(\text{immh:immb})$)</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>$\text{SEE Advanced SIMD modified immediate}$</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>(128-$\text{UInt}(\text{immh:immb})$)</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 \textit{SSHL}.

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{Scalar} and \textit{Vector}.

\textbf{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  | 1  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

\textbf{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  | Rm | 0  | 1  | 0  | 1  | 0  | 1  | Rn | Rd |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
```

\textbf{Assembler Symbols}

\texttt{<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**

```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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 SSSR.

Depending on the settings in the CPACR_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 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------- |
<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;V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>immh</th>
<th>&lt;T&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>2F</td>
</tr>
<tr>
<td>01xx 1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx 0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx 1</td>
<td>2D</td>
</tr>
</tbody>
</table>

<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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SRSRA

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

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & != & 0000 & \text{immh} & 0 & 0 & 1 & 1 & 0 & 1 & \text{Rn} & \text{Rd} \\
\text{U} & \text{immh} & 01 & 00
\end{array}
\]

Scalar

SRSRA \langle V \rangle \langle d \rangle, \langle V \rangle \langle n \rangle, \#\langle shift \rangle

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

if \text{immh}<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - \text{UInt}(\text{immh}:\text{immb});
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Vector

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & != & 0000 & \text{immh} & 0 & 0 & 1 & 1 & 0 & 1 & \text{Rn} & \text{Rd} \\
\text{U} & \text{immh} & 01 & 00
\end{array}
\]

Vector

SRSRA \langle Vd \rangle.\langle T \rangle, \langle Vn \rangle.\langle T \rangle, \#\langle shift \rangle

integer d = \text{UInt}(Rd);
integer n = \text{UInt}(Rn);

if \text{immh} == '0000' then SEE(asimdimm);
if \text{immh}<3>:Q == '10' then UNDEFINED;
integer esize = 8 << \text{HighestSetBit}(\text{immh});
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - \text{UInt}(\text{immh}:\text{immb});
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');

Assembler Symbols

\langle V \rangle \quad \text{Is a width specifier, encoded in “immh”:}
immh <V>

0xxx RESERVED
1xxx 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.
<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>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0001</td>
<td>0B</td>
</tr>
<tr>
<td>0001 1</td>
<td>16B</td>
</tr>
<tr>
<td>001x 0</td>
<td>04H</td>
</tr>
<tr>
<td>001x 1</td>
<td>08H</td>
</tr>
<tr>
<td>01xx 0</td>
<td>2S</td>
</tr>
<tr>
<td>01xx 1</td>
<td>4S</td>
</tr>
<tr>
<td>1xxx 0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx 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 right shift amount, in the range 1 to 64, encoded in “immh:immb”:

<table>
<thead>
<tr>
<th>immh</th>
<th>shift</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>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>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

**Operation**

CheckFPAdvSIMDEnabled64();
bias(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SSHL

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

| 31 30 29 28 27 26 25 24 23 22 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 0 1 | Rn  | Rd  |
| U    | R           | S      |

Scalar

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

| 31 30 29 28 27 26 25 24 23 22 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 | size 1 | Rm  | 0 1 0 0 0 1 | Rn  | Rd  |
| Q   | R           | S      |

Vector

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

<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>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>8B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>16B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>4H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>8H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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.
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>
<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>U</th>
<th>immh</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q</td>
<td>0</td>
</tr>
</tbody>
</table>

Vector

SSHLL{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>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>
Is the left shift amount, in the range 0 to the source 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>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>immb == '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 SSRSR.

Depending on the settings in the CPACR_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| 1 0| != 0000| immmb | 0 0| 0 0| 0 1| Rd |

**Scalar**

```
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 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | != 0000 | immmb | 0 0| 0 0| 0 1| Rd |

**Vector**

```
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”: 

Page 1141
<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>0xxx</td>
<td>x</td>
<td>SEE Advanced SIMD modified immediate</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
<td>8B</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>16B</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td>8H</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
<td>4S</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0111</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 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>0010</td>
<td>(32-UInt(immh:immb))</td>
</tr>
<tr>
<td>0011</td>
<td>(64-UInt(immh:immb))</td>
</tr>
<tr>
<td>0110</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

Operation

```
CheckFPAvSIMDEnabled64();
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 0  != 0000  immh  0 0 0 1 0 1  Rn  Rd

Scalar

SSRA <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 0 0 1 1 1 1 1 0  != 0000  immh  0 0 0 1 0 1  Rn  Rd

Vector

SSRA <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”:
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>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>(128-UInt(immh:immb))</td>
<td></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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>SEE_Advanced_SIMD_modified_immediate</td>
<td></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>

<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.
SSUBL, SSUBL2

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 \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
| Q | 0 | 1 | 1 | 0 | size | 1 | Rm | 0 | 0 | 1 | 0 | 0 | Rn | Rd | 0 | 1

Three registers, not all the same type

\texttt{SSUBL(2) \langle Vd\rangle.<Ta>, \langle Vn\rangle.<Tb>, \langle Vm\rangle.<Tb>}

integer d = \texttt{UINT(Rd)};
integer n = \texttt{UINT(Rn)};
integer m = \texttt{UINT(Rm)};

if size == '11' then UNDEFINED;
integer esize = 8 << \texttt{UINT(size)};
integer datasize = 64;
integer part = \texttt{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":

\begin{verbatim}
Q  2
| 0 | [absent] |
| 1 | [present] |
\end{verbatim}

\texttt{\langle Vd\rangle} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{\langle Ta\rangle} Is an arrangement specifier, encoded in "size":

\begin{verbatim}
size  <Ta>
| 00 | 8H |
| 01 | 4S |
| 10 | 2D |
| 11 | RESERVED |
\end{verbatim}

\texttt{\langle Vn\rangle} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\texttt{\langle Tb\rangle} Is an arrangement specifier, encoded in "size:Q":

\begin{verbatim}
size  Q  <Tb>
| 00 | 0 | 8B |
| 00 | 1 | 16B |
| 01 | 0 | 4H |
| 01 | 1 | 8H |
| 10 | 0 | 2S |
| 10 | 1 | 4S |
| 11 | x | RESERVED |
\end{verbatim}

\texttt{\langle Vm\rangle} 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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

Three registers, not all the same type

SSUBW(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":

<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

\[ \text{CheckFPAdvSIMDEnabled64()} \]
\[
\text{bits}(2 \times \text{datasize}) \ \text{operand1} = V[n];
\]
\[
\text{bits(\text{datasize})} \ \text{operand2} = Vpart[m, \text{part}];
\]
\[
\text{bits}(2 \times \text{datasize}) \ \text{result};
\]
\[
\text{integer element1;}
\]
\[
\text{integer element2;}
\]
\[
\text{integer sum;}
\]
\[
\text{for e = 0 to elements-1}
\]
\[
\text{element1} = \text{Int}(\text{Elem}[\text{operand1}, e, 2 \times \text{esize}], \text{unsigned});
\]
\[
\text{element2} = \text{Int}(\text{Elem}[\text{operand2}, e, \text{esize}], \text{unsigned});
\]
\[
\text{if sub_op then}
\]
\[
\text{sum} = \text{element1} - \text{element2};
\]
\[
\text{else}
\]
\[
\text{sum} = \text{element1} + \text{element2};
\]
\[
\text{Elem}[\text{result}, e, 2 \times \text{esize}] = \text{sum}<2 \times \text{esize}-1:0>;
\]
\[ V[d] = \text{result}; \]

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.
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 \textit{CPACR\_EL1, 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: \textbf{No offset} and \textbf{Post-index}

### No offset

\begin{verbatim}
<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>L</td>
</tr>
</tbody>
</table>
\end{verbatim}

**One register (opcode == 0111)**

\texttt{ST1 \{ <Vt>..<T> \}, [<Xn|SP]>}

**Two registers (opcode == 1010)**

\texttt{ST1 \{ <Vt>..<T>, <Vt2>..<T> \}, [<Xn|SP]>}

**Three registers (opcode == 0110)**

\texttt{ST1 \{ <Vt>..<T>, <Vt2>..<T>, <Vt3>..<T> \}, [<Xn|SP]>}

**Four registers (opcode == 0010)**

\texttt{ST1 \{ <Vt>..<T>, <Vt2>..<T>, <Vt3>..<T>, <Vt4>..<T> \}, [<Xn|SP]>}

\begin{verbatim}
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
\end{verbatim}

### Post-index

\begin{verbatim}
<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>L</td>
</tr>
</tbody>
</table>
\end{verbatim}
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

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!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

<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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>S</td>
<td>size</td>
<td>Rn</td>
<td></td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<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>
</tr>
</tbody>
</table>

**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

<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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>S</td>
<td>size</td>
<td>Rn</td>
<td></td>
<td>Rm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>R</td>
<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>
</tr>
</tbody>
</table>

Historical Note: This encoding was introduced in the Cortex®-M0 plus and Cortex®-M3 processors. It was then adopted by later Cortex®-M processor families to ensure compatibility.
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 == 00)

ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

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.
<int> 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 scale = UInt(opcode<2:1>);
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
    SetNotTagCheckedInstruction(!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_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 0| 0 0 0 0 0 0| 1 0 0 0| size | Rn | Rt |

L  
opcode

### Immediate offset (Rm == 11111)

ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>

### Register offset (Rm ! 11111)

ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 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 0| Rm | 1 0 0 0| size | Rn | Rt |

L  
opcode

### 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 0</td>
<td>08B</td>
<td></td>
</tr>
<tr>
<td>00 1</td>
<td>16B</td>
<td></td>
</tr>
<tr>
<td>01 0</td>
<td>04H</td>
<td></td>
</tr>
<tr>
<td>01 1</td>
<td>08H</td>
<td></td>
</tr>
<tr>
<td>10 0</td>
<td>02S</td>
<td></td>
</tr>
<tr>
<td>10 1</td>
<td>04S</td>
<td></td>
</tr>
<tr>
<td>11 0</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>11 1</td>
<td>02D</td>
<td></td>
</tr>
</tbody>
</table>

- **<Vt2>** Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
\(<\text{Xn}|\text{SP}?>\) Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

\(<\text{imm}>\) Is the 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>

\(<\text{Xm}>\) 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;
```
**Operation**

```c
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 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>]

```plaintext
t = UInt(Rt);
n = UInt(Rn);
m = integer UNKNOWN;
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 |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | | | | Rn | Rm | S | size | x | x | 0 | |

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>

\[
\begin{align*}
\text{integer } t & = \text{UInt}(Rt); \\
\text{integer } n & = \text{UInt}(Rn); \\
\text{integer } m & = \text{UInt}(Rm); \\
\text{boolean } wback & = \text{TRUE}; \\
\text{boolean } \text{tag\_checked} & = wback || n \neq 31;
\end{align*}
\]

**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 scale = UInt(opcode<2:1>);
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
  SetNotTagCheckedInstruction(!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];
    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

| 31 30 29 28 27 26 25 24 23 22 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   | 1   | 0   | 0   |
| size| Rn  | Rt  |

L  opcode

No offset

ST3 { <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   | 0   | 0   | 0   | 0   | 1   | 0   | 0   |
| size| Rm  | Rn  | Rt  |

L  opcode

Immediate offset (Rm == 11111)

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], [<imm>]

Register offset (Rm != 11111)

ST3 { <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

<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>
</tr>
</thead>
<tbody>
<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>

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as “Rt” plus 1 modulo 32.
Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 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>#24</td>
</tr>
<tr>
<td>1</td>
<td>#48</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

```java
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;
```

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
  SetNotTagCheckedInstruction(!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.
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 |
|----------------------------------|-----------------|---------------|
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 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 |
|----------------------------------|-----------------|---------------|
| 0 | Q | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | x | x | 1 | S | size | Rn | Rt |
| L | R | opcode |
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 scale = UInt(opcode<2:1>);
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<l> == '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
    SetNotTagCheckedInstruction(!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.

---

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \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**

| 0 | Q | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | size | Rt | Rn |
| L | opcode |

**No offset**

ST4 \{ <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**

<table>
<thead>
<tr>
<th>0</th>
<th>Q</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>size</th>
<th>Rt</th>
<th>Rn</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<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>
</tr>
</tbody>
</table>

**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 = 1; selem = 1;  // LD/ST1 (1 register)
  when '1000' rpt = 2; selem = 1;  // LD/ST1 (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
    SetNotTagCheckedInstruction(!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.
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| L  | R  | opcode |

8-bit (opcode == 001)


16-bit (opcode == 011 && size == 00)


32-bit (opcode == 101 && size == 00)


64-bit (opcode == 101 && S == 0 && size == 01)


```java
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  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| 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)

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 scale = UInt(opcode<2:1>);
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]

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
    SetNotTagCheckedInstruction(!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.
STNP (SIMD&FP)

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.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|opc| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|imm7|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|Rt2|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|Rn|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|Rt|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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

```plaintext
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
    SetNotTagCheckedInstruction(!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

<table>
<thead>
<tr>
<th>opc</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>opc</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>opc</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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;
bv64 offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
Operation

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTEExt() then
  SetNotTagCheckedInstruction(!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>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>
</table>

**opc**

**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;
bias(64) offset = SignExtend(imm9, 64);

### Pre-index

<table>
<thead>
<tr>
<th>size</th>
<th>1</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>1</th>
<th>1</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

**opc**
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 = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bets(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 | 0 | 1 | x | 0 | imm12 | Rn | Rt |
|---------------------------------------------|------|-----|-----|-----|----|---|---|------|-----|-----|
| opc | 8-bit (size == 00 && opc == 00) | STR <Bt>, [<Xn|SP>{, #<pimm}>]

16-bit (size == 01 && opc == 00)

STR <Ht>, [<Xn|SP>{, #<pimm}>]

32-bit (size == 10 && opc == 00)

STR <St>, [<Xn|SP>{, #<pimm}>]

64-bit (size == 11 && opc == 00)

STR <Dt>, [<Xn|SP>{, #<pimm}>]

128-bit (size == 00 && opc == 10)

STR <Qt>, [<Xn|SP>{, #<pimm}>]

boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bets(64) offset = LSL(ZeroExtend(imm12, 64), scale);
Assembler Symbols

<\texttt{Bt}> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<\texttt{Dt}> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<\texttt{Ht}> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<\texttt{Qt}> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<\texttt{St}> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<\texttt{Xn}\mid\texttt{SP}> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<\texttt{simm}> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<\texttt{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);
```

STR (immediate, SIMD&FP)
Operation

if `HaveMTEExt()` then
    SetNotTagCheckedInstruction(!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.

| 31 30 29 28 27 26 25 24 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 | 0 | x | 0 | 1 | Rm | option | S | 1 | 0 | Rn | Rt |

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>}}]

```plaintext
ingTEGER 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>

<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 tag_checked = memop != MemOp_PREFETCH;
```
**Operation**

```c
bits(64) offset = ExtendReg(m, extend_type, shift);
if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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>imm9</th>
<th>0 0</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td>1 1 1 1 1</td>
<td>0 0</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

if HaveMTEExt() then
    SetNotTagCheckedInstruction(!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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 1 0 | size | 1 | Rm | 1 0 0 0 0 1 | Rn | Rd |
```

Scalar

```
integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
datasize = esize;
elements = 1;
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 0 1 0 1 1 1 0 | size | 1 | Rm | 1 0 0 0 0 1 | Rn | Rd |
```

Vector

```
integer d = UInt(Rd);
n = UInt(Rn);
m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
datasize = esize;
datasize = if Q == '1' then 128 else 64;
datasize = datasize DIV esize;
sub_op = (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>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;
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.

```
<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>Q</td>
<td>0</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>1</td>
<td>1</td>
<td>0</td>
<td>0</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>
```

Three registers, not all the same type

SUBHN(2) <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```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

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

\texttt{CheckFPAdvSIMDEnabled64();}
\texttt{bits(2*datasize) operand1 = V[n];}
\texttt{bits(2*datasize) operand2 = 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{\hspace{1em}element1 = Elem[operand1, e, 2*esize];}
\texttt{\hspace{1em}element2 = Elem[operand2, e, 2*esize];}
\texttt{\hspace{1em}if sub\_op then}
\texttt{\hspace{2em}sum = element1 - element2;}
\texttt{\hspace{1em}else}
\texttt{\hspace{2em}sum = element1 + element2;}
\texttt{\hspace{1em}sum = sum + round\_const;}
\texttt{\hspace{1em}Elem[result, e, esize] = sum<2*esize-1:esize>;
\texttt{Vpart[d, part] = 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}
**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_A64ISAR0_EL1.I8MM 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>0</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>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>
</tr>
</tbody>
</table>

**US**

**Vector**

SUDOT <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.
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;
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>&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.
Is the number of the 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 "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 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | Q  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | Rm | 0  | len | 0  | 0  | 0  | Rn |   |   |   |   |   |   |   |   |   | Rd |

**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>

```plaintext
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

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | Rm | 0  | len | 1  | 0  | 0  | Rn |   |   |   |   | 1  | 64 |

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>
```

```plaintext
generate assemble code
```

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":
    
    | Q | <Ta> |
    |---|------|
    | 0 | 8B   |
    | 1 | 16B  |
- `<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

\textbf{CheckFPAdvSIMDEnabled64();} \\
bits(datasize) indices = \texttt{V}[m]; \\
bits(128*regs) table = \texttt{Zeros}(); \\
bits(datasize) result; \\
integer index;

// Create table from registers 
for \( i = 0 \) to \( \text{regs}-1 \) 
\quad \text{table}::<128*i+127:128*i> = \texttt{V}[n]; 
\quad \text{n} = (\text{n} + 1) \text{ MOD } 32;

result = if is tbl then \texttt{Zeros}() else \texttt{V}[d]; 
for \( i = 0 \) to \( \text{elements}-1 \) 
\quad \text{index} = \texttt{UInt}(\texttt{Elem}[indices, i, 8]); 
\quad \text{if index} < 16 * \text{regs} \text{ then} 
\quad \quad \texttt{Elem}[\text{result, i, 8}] = \texttt{Elem}[\text{table, index, 8}];
\texttt{V}[d] = \text{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.
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.

![Example Figure]

Depending on the settings in the CPACR_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 | Rd 0 0 1 0 1 0 | Rn 0
```

Advanced SIMD

TRN1 <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;
integer part = 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>

<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 = \textit{V}[n];
bits(datasize) operand2 = \textit{V}[m];
bits(datasize) result;

for p = 0 to pairs-1
    \textit{Elem}[result, 2*p+0, esize] = \textit{Elem}[operand1, 2*p+part, esize];
    \textit{Elem}[result, 2*p+1, esize] = \textit{Elem}[operand2, 2*p+part, esize];
\textit{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 $2 \times 2$ matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where $Q = 0$.

![Example figure showing TRN1 and TRN2 operations]

Depending on the settings in the `CPACR_EL1`, `CPTR_EL2`, and `CPTR_EL3` registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

TRN2 `<Vd>..<T>`, `<Vn>..<T>`, `<Vm>..<T>`

integer $d = \text{UInt}(Rd)$;
integer $n = \text{UInt}(Rn)$;
integer $m = \text{UInt}(Rm)$;

if $\text{size}:Q == '110'$ then UNDEFINED;
integer $\text{esize} = 8 \ll \text{UInt}(\text{size})$;
integer $\text{datasize} = \text{if } Q == '1' \text{ then } 128 \text{ else } 64$;
integer $\text{elements} = \text{datasize} \div \text{esize}$;
integer $\text{part} = \text{UInt}(\text{op})$;
integer $\text{pairs} = \text{elements} \div 2$;

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>

`<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 = V[n];
bits(datasize) operand2 = V[m];
bits(datasize) result;
for p = 0 to pairs-1
    \texttt{Elem[result, 2*p+0, esize]} = \texttt{Elem[operand1, 2*p+part, esize]};
    \texttt{Elem[result, 2*p+1, esize]} = \texttt{Elem[operand2, 2*p+part, esize]};
V[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 registers.
  - The values of the NZCV flags.
- The response of this instruction to 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.

Three registers of the same type

UABA `<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

`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.

Three registers, not all the same type

UABAL{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 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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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();
bv[datasize] operand1 = Vpart[n, part];
bv[datasize] operand2 = Vpart[m, part];
bv[2*datasize] result;
integer element1;
integer element2;
bv[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.

Three registers of the same type

UABD <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>&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.
**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 | 1 | 0 | size | 1 | Rm | 0 | 1 | 1 | 1 | 0 | 0 | Rn | Rd | op |

Three registers, not all the same type

UABDL[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 accumulate = (op == '0');  
boolean unsigned = (U == '1');  
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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.
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.

```
| 31 30 29 28 27 26 25 24 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:Q          | Rd              | Rn              | op              |
```

**Vector**

UADALP <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 Q</th>
<th>&lt;Ta&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>4H</td>
</tr>
<tr>
<td>00 1</td>
<td>8H</td>
</tr>
<tr>
<td>01 0</td>
<td>2S</td>
</tr>
<tr>
<td>01 1</td>
<td>4S</td>
</tr>
<tr>
<td>10 0</td>
<td>1D</td>
</tr>
<tr>
<td>10 1</td>
<td>2D</td>
</tr>
<tr>
<td>11 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 Q</th>
<th>&lt;Tb&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>8B</td>
</tr>
<tr>
<td>00 1</td>
<td>16B</td>
</tr>
<tr>
<td>01 0</td>
<td>4H</td>
</tr>
<tr>
<td>01 1</td>
<td>8H</td>
</tr>
<tr>
<td>10 0</td>
<td>2S</td>
</tr>
<tr>
<td>10 1</td>
<td>4S</td>
</tr>
<tr>
<td>11 x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
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>;
    Elem[result, e, 2*esize] = 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.
**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.

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

Three registers, not all the same type

UADDL\(2\) \(<Vd>\), \(<Ta>\), \(<Vn>\).\(<Tb>\)

```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 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) 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 \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.

**Assembler Symbols**

\texttt{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\texttt{<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>

\texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.

\texttt{<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>

```
31 30 29 28 27 26 25 24 23 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 0 0 0 0 0 0 0 1 0 1 0 Rd
   Rn
U op
```

Vector

UADDLP \texttt{<Vd>.<Ta>, <Vn>.<Tb>}

```
n = UInt(Rn);
d = UInt(Rd);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
datasize = if Q == '1' then 128 else 64;
elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```
Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

result = if acc then V[d] else Zeros();
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>;
    Elem[result, e, 2*esize] = 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.

<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 0 1 1 0</td>
<td>1 1 0 0 0 0 0 0 1 1 1 0</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

**Advanced SIMD**

UADDLV `<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.
**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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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 | 0 | 0 | 1 | 0 | 0 | Rn | | Rd | | o1 |

**Three registers, not all the same type**

**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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</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>8H</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>16H</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2S</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2H</td>
</tr>
<tr>
<td>11</td>
<td>x</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Operation

```c
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**

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 | 1 | 0 | !|= 0000 | immh | 1 | 1 | 1 | 0 | 0 | 1 | Rn  |   |   |   |   |   |   |   |   |   |   | Rd |
```

Scalar

```
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 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | !|= 0000 | immh | 1 | 1 | 1 | 0 | 0 | 1 | Rn  |   |   |   |   |   |   |   |   |   |   | Rd |
```

Vector

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000x' || (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 = FPRoundingMode(FPCR);
```

**Assembler Symbols**

- `<V>`: Is a width specifier, encoded in “immh”:
<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_modifiedImmediate</td>
</tr>
<tr>
<td>0001</td>
<td>x</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01lx</td>
<td>0</td>
<td>4H</td>
</tr>
<tr>
<td>01lx</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>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>000x</td>
<td>SEE_Advanced_SIMD_modifiedImmediate</td>
</tr>
<tr>
<td>0001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01lx</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];
bfindatastize) result;
bfindatastize) element;

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)

<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>
<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>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<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>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>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**UCVTF <Hd>, <Hn>**

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

<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>
<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>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<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>sz</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>1</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>
</tr>
</tbody>
</table>

**UCVTF <V>d>, <V<n>**

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)

<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>
<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>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<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>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>1</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>
</tr>
</tbody>
</table>
Vector half precision

UCVTF <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

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>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 vector 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 vector 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;
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;
```
**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>sf</th>
<th>0 0 1 1 1 0</th>
<th>ftype</th>
<th>0 0 0 1 1</th>
<th>scale</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>
```

*sf* Mode
*ftype* Floating-point type
*scale* Scale factor
*Rn* General-purpose source register
*Rd* SIMD&FP destination register
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;
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();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = FixedToFP(intval, fracbits, TRUE, FPCR, rounding);
V[d] = fltval;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.
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

```c
CheckFPAdvSIMDEnabled64();

bits(fltsize) fltval;
bits(intsize) intval;

intval = X[n];
fltval = 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_AA64ISARO_EL1.DP indicates whether this instruction is supported.

Vector
(Armv8.2)

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>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 element index, encoded in the "H:L" fields.
Operation

CheckFPAdvSIMDEnabled64();
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 A64ISAR0_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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| U  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | size | 0  | Rm | 1  | 0  | 0  | 1  | 0  | 1  | Rn | 1  | 0  | 0  | 0  | 1  | 0  | 1  | Rd |

### Vector

**UDOT** `<Vd>`, `<Va>`, `<Vn>`, `<Vb>`

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": |
| **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

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;
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 0  | 0  | 1  | Rn | Rd |
| U  |

Three registers of the same type

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.

Three registers of the same type

UHSUB <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 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.
**UMAX**

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | Rm | 0  | 1  | 0  | 0  | 1  | Rn | Rd |

**Three registers of the same type**

UMAX `<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;
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.
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 \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.

Three registers of the same type

\texttt{UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>}

```plaintext
\begin{verbatim}
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');
\end{verbatim}
```

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":

```plaintext
\begin{array}{c|c}
\text{size} & \text{T} \\
\hline
00 & 8B \\
00 & 16B \\
01 & 4H \\
01 & 8H \\
10 & 2S \\
10 & 4S \\
11 & x \\
\end{array}
```

- \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

```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;
```

Operational information

If PSTATE.DIT is 1:

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

<table>
<thead>
<tr>
<th>Rn</th>
<th>Rd</th>
<th>size</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0010</td>
<td>010</td>
</tr>
</tbody>
</table>

### Advanced SIMD

**UMAXV <V>, <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');
boolean min = (op == '1');
```

### Assembler Symbols

- `<V>` Is the destination width specifier, encoded in “size”:
  - `size`<V>
    - 00: B
    - 01: H
    - 10: S
    - 11: RESERVED

- `<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”:
  - `size` Q `<T>`
    - 00 0: 8B
    - 00 1: 16B
    - 01 0: 4H
    - 01 1: 8H
    - 10 0: RESERVED
    - 10 1: 4S
    - 11: x RESERVED

### Operation

```
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>;
```
Operational information

If PSTATE.DIT is 1:

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

- The response of this instruction to 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.

Three registers of the same type

UMIN <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.
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.

Three registers of the same type

UMINP <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.
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.

![Register Diagram]

**Advanced SIMD**

UMINV `<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');
boolean min = (op == '1');

**Assembler Symbols**

<table>
<thead>
<tr>
<th><code>&lt;V&gt;</code></th>
<th>Is the destination width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td><code>&lt;V&gt;</code></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>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”: |
| size  | Q | `<T>` |
| 00    | 0 | 8B    |
| 00    | 1 | 16B   |
| 01    | 0 | 4H    |
| 01    | 1 | 8H    |
| 10    | 0 | RESERVED |
| 10    | 1 | 4S    |
| 11    | x | RESERVED |

**Operation**

```c
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>;
```
Operational information

If PSTATE.DIT is 1:

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

• The response of this instruction to 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.

Vector

UMLAL[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>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”: 
Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

<table>
<thead>
<tr>
<th>size</th>
<th>Q</th>
<th>&lt;Vm&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>

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
CheckFPAvSIMDEnabled64();
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.

```
0 | Q | 1 | 0 | 1 | 1 | 0 | size | 1 | Rm | 1 | 0 | 0 | 0 | 0 | 0 | Rn | Rd |
```

### Three registers, not all the same type

**UMLAL**<sub>2</sub>  
\(<V_d>, <Ta>, <V_n>, <T_b>\), \(<V_m>, <T_b>\)

```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>

\(<V_d>\)  
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
```

\(<V_n>\)  
Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\(<T_b>\)  
Is an arrangement specifier, encoded in "size:Q":

```
size  Q  <T_b>
00    0    8B
00    1    16B
01    0    4H
01    1    8H
10    0    2S
10    1    4S
11    X    RESERVED
```

\(<V_m>\)  
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.
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.

### Vector

UMLSL(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>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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | 1  | 0  | 0  | 0  | Rd |

### Three registers, not all the same type

UMLSL(2) <Vd>, <Ta>, <Vn>, <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”:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>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.
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 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  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  |
| U  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rm |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rn |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Rd |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Vector

UMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

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

CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];

V[d] = MatMulAdd(addend, operand1, operand2, TRUE, TRUE);

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UMOV

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 \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 instruction is used by the alias \textit{MOV (to general)}.

\begin{center}
\begin{tabular}{ccccccccccccccccccccccc}
\hline
0 & Q & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & imm5 & 0 & 0 & 1 & 1 & 1 & Rn & Rd
\end{tabular}
\end{center}

\textbf{32-bit (Q == 0)}

\texttt{UMOV <Wd>, <Vn>.<Ts>[<index>]}

\textbf{64-reg, UMOV-64-reg (Q == 1 &amp; imm5 == x1000)}

\texttt{UMOV <Xd>, <Vn>.<Ts>[<index>]}

integer d = \texttt{UInt}(Rd);
integer n = \texttt{UInt}(Rn);

integer size;
\textbf{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 = \texttt{UInt}(imm5<4:size+1>);
integer esize = 8 \ll size;
integer datasize = if Q == '1' then 64 else 32;

\textbf{Assembler Symbols}

\texttt{<Wd>} Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
\texttt{<Xd>} Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
\texttt{<Vn>} Is the name of the SIMD&FP source register, encoded in the "Rn" field.
\texttt{<Ts>} For the 32-bit variant: is an element size specifier, encoded in "imm5":

\begin{center}
\begin{tabular}{cc}
\texttt{imm5} & \texttt{<Ts>} \\
xx000 & RESERVED \\
xxxx1 & B \\
xxx10 & H \\
xx100 & S \\
\end{tabular}
\end{center}

For the 64-reg, UMOV-64-reg variant: is an element size specifier, encoded in "imm5":

\begin{center}
\begin{tabular}{cc}
\texttt{imm5} & \texttt{<Ts>} \\
x0000 & RESERVED \\
xxxxx1 & RESERVED \\
xxx10 & RESERVED \\
xx100 & RESERVED \\
x1000 & D \\
\end{tabular}
\end{center}

\texttt{<index>} For the 32-bit variant: is the element index encoded in "imm5":

\begin{center}
\begin{tabular}{cc}
\texttt{imm5} & \texttt{<Ts>} \\
x0000 & RESERVED \\
xxxxx1 & RESERVED \\
xxx10 & RESERVED \\
xx100 & RESERVED \\
x1000 & D \\
\end{tabular}
\end{center}
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 == 'x1000'</td>
</tr>
<tr>
<td>MOV (to general)</td>
<td>imm5 == 'xx100'</td>
</tr>
</tbody>
</table>

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bits(idxdsize) 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.

<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>L</td>
<td>M</td>
<td>Rm</td>
<td>1</td>
<td>0</td>
<td>1</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>

Vector

\( \text{UMULL}(2) <Vd>, <Ta>, <Vn>, <Vm>, <Ts>[<index>] \)

integer idxdsizesize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
  when '01' index = \( \text{UInt}(H:L:M) \); Rmhi = '0';
  when '10' index = \( \text{UInt}(H:L) \); Rmhi = M;
  otherwise UNDEFINED;

integer d = \( \text{UInt}(Rd) \);
integer n = \( \text{UInt}(Rn) \);
integer m = \( \text{UInt}(Rmhi:Rm) \);

integer esize = 8 << \( \text{UInt}(\text{size}) \);
integer datasize = 64;
integer part = \( \text{UInt}(Q) \);
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>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>[absent]</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) 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)

Three registers, not all the same type

**UMULL{2}** <Vd>, <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 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

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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

<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>0 0 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalar

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

<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>0 0 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector

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

<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>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.
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 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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 \textit{UQSHL}.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit \textit{FPSR}.QC is set.

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: \textbf{Scalar} and \textbf{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 0</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>0 1 0 1 1 1</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>S</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>0</td>
</tr>
<tr>
<td>U</td>
</tr>
<tr>
<td>0 1 0 1 1 1</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>S</td>
</tr>
</tbody>
</table>

**Assembler Symbols**

\textit{<V>\textgreater;\textless;d\textgreater;, \textless;V\textgreater;\textless;n\textgreater;, \textless;V\textgreater;\textless;m\textgreater;}

integer d = \texttt{UInt}(Rd);\ninteger n = \texttt{UInt}(Rn);\ninteger m = \texttt{UInt}(Rm);\ninteger esize = 8 \ll \texttt{UInt}(size);\ninteger datasize = esize;\ninteger elements = 1;\nboolean unsigned = (U == '1');\nboolean rounding = (R == '1');\nboolean saturating = (S == '1');\nif S == '0' && size != '11' then UNDEFINED;

\textit{<Vd>.\textless;T\textgreater;, \textless;Vn>.\textless;T\textgreater;, \textless;Vm>.\textless;T\textgreater;}

integer d = \texttt{UInt}(Rd);\ninteger n = \texttt{UInt}(Rn);\ninteger m = \texttt{UInt}(Rm);\nif size:Q == '110' then UNDEFINED;\ninteger esize = 8 \ll \texttt{UInt}(size);\ninteger datasize = if Q == '1' then 128 else 64;\ninteger elements = datasize DIV esize;\nboolean unsigned = (U == '1');\nboolean rounding = (R == '1');\nboolean saturating = (S == '1');

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>;
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Unsigned 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, 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 \textit{UQSHRN}.

The \textit{UQRSHRN} instruction writes the vector to the lower half of the destination register and clears the upper half, while the \textit{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 \textit{FPSR}.QC is set.

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{Scalar} and \textit{Vector}

\textbf{Scalar}

\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  | ! = 0000 | immh | 1  | 0  | 0  | 1  | 1  | Rn  | Rd  |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\end{verbatim}

\textbf{Vector}

\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  | Q | 1  | 0  | 1  | 1  | 1  | 0  | ! = 0000 | immh | 1  | 0  | 0  | 1  | 1  | Rn  | Rd  |
| U  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| op |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

\end{verbatim}

\textbf{Scalar}

\textbf{UQRSHRN} \langle Vb \rangle \langle d \rangle, \langle Va \rangle \langle n \rangle, \langle \#shift \rangle

\begin{verbatim}

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');

\end{verbatim}

\textbf{Vector}

\textbf{UQRSHRN2} \langle \textit{Vd} \rangle.\langle \textit{Tb} \rangle, \langle \textit{Vn} \rangle.\langle \textit{Ta} \rangle, \langle \#shift \rangle

\begin{verbatim}

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 elements = datasize DIV esize;
integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');

\end{verbatim}
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>25</td>
</tr>
<tr>
<td>01xx</td>
<td>1</td>
<td>45</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;
**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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | != 0000 | immh | 0  | 1  | 1  | 1  | 0  | 1  | Rd | Rn |

#### Scalar

**UQSHL <V><d>, <V><n>, #<shift>**

```plaintext
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  | Q  | 1  | 0  | 1  | 1  | 1  | 0  | != 0000 | immh | 0  | 1  | 1  | 1  | 0  | 1  | Rd | Rn |

#### Vector

**UQSHL (immediate)**
Vector

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>
<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 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":
### 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;
```

---

**Operation**

<table>
<thead>
<tr>
<th>immh</th>
<th>Description</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>

---

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

---

UQSHL (immediate)  Page 1281
**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

```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 1 1 1 1 0   | size 1          | Rm              | 0 1 0           | 1 1             |
|                |                 | Rd              | Rd              |
```

### Vector

```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 Q 1 0 1 1 1 0 | size 1          | Rm              | 0 1 0           | 0 1 1           |
|                 |                 | Rd              | Rd              |
```

### Assembler Symbols

- `<V>` is a width specifier, encoded in “size”:
<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>&lt;V&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>H</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>S</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td>11</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;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 0 | immh 1 0 0 1 0 1 | Rn Rd |
| U op |

#### Scalar

**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 0 1 0 1 1 1 1 0 | immh 1 0 0 1 0 1 | Rn Rd |
| U op |
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');
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.
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**

```plaintext
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;
```

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 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 0 1 0 1 1 | Rn | Rd |
```

Scalar

UQSUB <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 Q 1 0 1 1 1 0 | size | 1 | Rm | 0 0 1 0 1 1 | Rn | Rd |
```

Vector

UQSUB <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.
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**

```cpp
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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

| 31 30 29 28 27 26 25 24 23 22 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 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | Rn | Rd | U |

Scalar

UQXTN <Vb><d>, <Va><n>

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 | Q | 1 | 0 | 1 | 1 | 1 | 0 | size | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | Rn | Rd | U |

Vector

UQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>

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".
<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>0H</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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

![Register Format](image)

**Vector**

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**

<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>

`<Vd>` Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

`<T>` Is an arrangement specifier, encoded in "sz:Q":

`<Vn>` Is the name of the SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```plaintext
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

### Assembler Symbols

<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>

**<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

```plaintext
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 30 29 28 27 26 25 24 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>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Scalar

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 30 29 28 27 26 25 24 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>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>
</tbody>
</table>

#### Vector

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><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.
<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;
```

Copyright © 2010-2019 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

| 31 30 29 28 27 26 25 24 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      | o1 o0     |

Scalar 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

| 31 30 29 28 27 26 25 24 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      | o1 o0     |

Vector URSHR <V>d.<T>, <V>n.<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”:
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>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1xxx</td>
<td>D</td>
</tr>
</tbody>
</table>

<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;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
type 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2.1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**URSQRTE**

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 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  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

**Vector**

URSQRTE <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] = UnsignedRSqrtEstimate(element);
V[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 USRA.

Depending on the settings in the CPACR_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 | Rd |
U | immb | o1 o0 | Rn |

Scalar

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 0 1 0 1 1 1 1 0 | != 0000 | immh | 0 0 1 1 0 1 | Rd |
U | immb | o1 o0 | Rn |

Vector

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”: 
### immh <V>

<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>&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>1xx</td>
<td>0 RESERVED</td>
</tr>
<tr>
<td>1xx</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>1xx</td>
<td>(128-UInt(immh:immb))</td>
</tr>
</tbody>
</table>

### Operation

```c
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n];
bids(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    | Q  | 0  | 1  | 1  | 1  | 1  | 0  | L  | M  | Rm |    | 1  | 1  | 1  | 1  | H  |    | 0  |   |    |    |    |    |    |    |    |    |    |
| US |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**Vector**

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 quadruplet of four 8-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(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)**

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

**Vector**

USDOT `<Vd>`.<`Ta`>, `<Vn>`.<`Tb`>, `<Vm>`.<`Tb`>

```markdown
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":
  ```markdown
<table>
<thead>
<tr>
<th>Q</th>
<th>Ta</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":
  ```markdown
<table>
<thead>
<tr>
<th>Q</th>
<th>Tb</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**

```markdown
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3; Build timestamp: 2019-09-27T18:00
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 URSHL.

Depending on the settings in the CPACR_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 | 0 1 | Rn | Rd |
| U | R | S |

Scalar

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 |
|--------------------|-----------------|-----------------|-----------------|
| 0 | Q | 1 0 1 1 1 0 | size | 1 | Rm | 0 1 0 0 | 0 1 | Rn | Rd |
| U | R | S |

Vector

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>
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;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 |
|-----------------------------|-----------------------------|-----------------------------|
| 0 1 0 1 1 1 0 | != 0000 | immh | 1 0 1 0 0 1 | Rn | Rd |
| U mmh |

**Vector**

USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

```
normal integer d = UInt(Rd);
normal integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
normal integer esize = 8 << HighestSetBit(immh);
normal integer datasize = 64;
normal integer part = UInt(Q);
normal integer elements = datasize DIV esize;

normal integer shift = UInt(immh:immb) - esize;
normal boolean unsigned = (U == '1');
```

**Assembler Symbols**

<table>
<thead>
<tr>
<th>0</th>
<th>1</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>

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>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>
Is the left shift amount, in the range 0 to the source 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>RESERVED</td>
</tr>
</tbody>
</table>

**Alias Conditions**

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>UXTL, UXTL2</td>
<td>immb == '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.
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 URSHR.

Depending on the settings in the CPACR_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 | != 0000 | immmb | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
```

Scalar

USHR \(<V>\langle d\rangle, \langle V\rangle\langle n\rangle, \#\langle shift\rangle\)

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

```
| 0 | Q | 1 | 0 | 1 | 1 | 1 | 1 | != 0000 | immmb | 0 | 0 | 0 | 0 | 1 | Rn | Rd |
```

Vector

USHR \(<Vd>.\langle T\rangle, \langle Vn\rangle.\langle T\rangle, \#\langle shift\rangle\)

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”:
<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>Vd</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xxx</td>
<td>0B</td>
<td>0xxx</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0xxx</td>
<td>16B</td>
<td>0xx</td>
<td>0</td>
</tr>
<tr>
<td>0xl</td>
<td>0</td>
<td>04H</td>
<td>1</td>
</tr>
<tr>
<td>0xl</td>
<td>1</td>
<td>08H</td>
<td>1</td>
</tr>
<tr>
<td>0xx</td>
<td>0</td>
<td>02S</td>
<td>0</td>
</tr>
<tr>
<td>0xx</td>
<td>1</td>
<td>04S</td>
<td>1</td>
</tr>
<tr>
<td>1xxx</td>
<td>0</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>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.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in "immh:immb":

 immh | shift |
<table>
<thead>
<tr>
<th></th>
<th></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":

 immh | shift |
<table>
<thead>
<tr>
<th></th>
<th></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>00lx</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();
b bits(datasize) operand = V[n];
b bits(datasize) operand2;
b 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_EL1.18MM 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  | 0  | 0  | 1  | 1  | Rn | Rd |

Vector

USMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

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

CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n];
bits(128) operand2 = V[m];
bits(128) addend = V[d];
V[d] = MatMulAdd(addend, operand1, operand2, TRUE, FALSE);

Copyright © 2010-2019 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

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

Scalar

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

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

Vector

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.
Is the number of the 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 "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 SIMD&FP source register, encoded in the "Rn" field.

**Operation**

```c
CheckFPAdvSIMDEnabled64();
bias(datasize) operand = V[n];
bias(datasize) result;

bias(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 USRSA.

Depending on the settings in the CPACR_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

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

USRA <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”:
<immh> 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>T</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 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;
```

**Operational information**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 \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{center}
\begin{tabular}{cccccccccccccccc}
0 & Q & 1 & 0 & 1 & 1 & 0 & \textbf{size} & 1 & \textbf{Rm} & 0 & 0 & 1 & 0 & 0 & 0 & \textbf{Rn} & \textbf{Rd} & 0 & 1 \\
\end{tabular}
\end{center}

Three registers, not all the same type

\begin{verbatim}
USUBL(2) <Vd>,<Ta>, <Vn>,<Tb>, <Vm>

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dat-size = 64;
integer part = UInt(Q);
integer elements = dat-size DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
\end{verbatim}

**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{center}
\begin{tabular}{c|c}
Q & 2 \\
0 & [absent] \\
1 & [present] \\
\end{tabular}
\end{center}

\textbf{<Vd>} Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

\textbf{<Ta>} Is an arrangement specifier, encoded in "size":

\begin{center}
\begin{tabular}{c|c}
\textbf{size} & \textbf{<Ta>} \\
00 & 8H \\
01 & 4S \\
10 & 2D \\
11 & RESERVED \\
\end{tabular}
\end{center}

\textbf{<Vn>} Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

\textbf{<Tb>} Is an arrangement specifier, encoded in "size:Q":

\begin{center}
\begin{tabular}{c|c|c}
\textbf{size} & \textbf{Q} & \textbf{<Tb>} \\
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & RESERVED \\
\end{tabular}
\end{center}

\textbf{<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.
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.

Three registers, not all the same type

USUBW{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

<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

\[
\text{CheckFPAdvSIMDEnabled64}();
\]
\[
\text{bits}(2 \times \text{datasize}) \text{ operand1} = V[n];
\]
\[
\text{bits}(\text{datasize}) \text{ operand2} = Vpart[m, \text{part}];
\]
\[
\text{bits}(2 \times \text{datasize}) \text{ result};
\]
\[
\text{integer element1};
\]
\[
\text{integer element2};
\]
\[
\text{integer sum};
\]
\[
\text{for e = 0 to elements-1}
\]
\[
\text{element1} = \text{Int}(\text{Elem(operand1, e, } 2 \times \text{esize}], \text{unsigned});
\]
\[
\text{element2} = \text{Int}(\text{Elem(operand2, e, esize}], \text{unsigned});
\]
\[
\text{if sub_op then}
\]
\[
\text{sum} = \text{element1} - \text{element2};
\]
\[
\text{else}
\]
\[
\text{sum} = \text{element1} + \text{element2};
\]
\[
\text{Elem(result, e, } 2 \times \text{esize}] = \text{sum}<2 \times \text{esize}-1:0>;
\]
\[
V[d] = \text{result};
\]

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.
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.

Depending on the settings in the CPACR_EL1, CPTR_EL2, and CPTR_EL3 registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Advanced SIMD

UZP1 <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;
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();`

```cpp
bits(datasize) operandl = V[n];
bits(datasize) operandh = V[m];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandsl;
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.

```
<table>
<thead>
<tr>
<th>A2</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A3</td>
<td>A7</td>
<td>A6</td>
<td>A5</td>
<td>A4</td>
<td>A3</td>
</tr>
</tbody>
</table>
```

UZP1.8, doubleword

```
<table>
<thead>
<tr>
<th>B6</th>
<th>B4</th>
<th>B2</th>
<th>B0</th>
<th>A6</th>
<th>A4</th>
<th>A2</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>B5</td>
<td>B3</td>
<td>B1</td>
<td>B0</td>
<td>A7</td>
<td>A5</td>
<td>A3</td>
<td>A1</td>
</tr>
</tbody>
</table>
```

UZP2.8, doubleword

Depending on the settings in the CPACR_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  0  1  1  1  0  size 0  0  1  0  1  1  0  Rm 0  1  0  1  1  0  Rn  Rd
```

Advanced SIMD

UZP2 <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;
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”:

```
size   Q
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

```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.
### XAR

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 **ARMv8.2-SHA** 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  | 1  | 0  | 0  | Rm | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

#### Advanced SIMD

XAR `<Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>`

```plaintext
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.
- `<imm6>` Is a rotation right, encoded in "imm6".

#### Operation

```plaintext
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m];
bits(128) Vn = V[n];
bits(128) tmp;
tmp = Vn EOR Vm;
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 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Q | 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  | 0  | 0  |
| Rd|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Vector

\[
\text{XTN}(2) \quad <\text{Vd}>, <\text{Tb}> , <\text{Vn}>, <\text{Ta}>
\]

\[
\begin{array}{ll}
\text{integer } d & = \text{UInt}(\text{Rd}); \\
\text{integer } n & = \text{UInt}(\text{Rn}); \\
\end{array}
\]

\[
\text{if } \text{size} \text{==} '11' \text{then UNDEFINED;} \\
\text{integer } \text{esize} & = 8 \ll \text{UInt(size)}; \\
\text{integer } \text{datasize} & = 64; \\
\text{integer } \text{part} & = \text{UInt}(Q); \\
\text{integer } \text{elements} & = \text{datasize} \text{DIV esize}; \\
\]

Assembler Symbols

\[
\begin{array}{ll}
\text{2} & \text{Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in "Q";} \\
\end{array}
\]

\[
\begin{array}{lll}
\text{Q} & \text{2} \\
\hline
0 & \text{[absent]} \\
1 & \text{[present]} \\
\end{array}
\]

\[
<\text{Vd}> \quad \text{Is the name of the SIMD&FP destination register, encoded in the "Rd" field.} \\
<\text{Tb}> \quad \text{Is an arrangement specifier, encoded in "size:Q";} \\
\begin{array}{lll}
\text{size} & \text{Q} & <\text{Tb}> \\
\hline
00 & 0 & 8B \\
00 & 1 & 16B \\
01 & 0 & 4H \\
01 & 1 & 8H \\
10 & 0 & 2S \\
10 & 1 & 4S \\
11 & x & \text{RESERVED} \\
\end{array}
\]

\[
<\text{Vn}> \quad \text{Is the name of the SIMD&FP source register, encoded in the "Rn" field.} \\
<\text{Ta}> \quad \text{Is an arrangement specifier, encoded in “size”:} \\
\begin{array}{ll}
\text{size} & <\text{Ta}> \\
\hline
00 & 8H \\
01 & 4S \\
10 & 2D \\
11 & \text{RESERVED} \\
\end{array}
\]
Operation

```c
CheckFPAdvSIMDEnabled64();
bhits(2*datasize) operand = \textbf{V}[n];
bhits(datasize) result;
bhits(2*esize) element;

for e = 0 to elements-1
    \textbf{elem} = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = \textbf{elem}<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 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 ZIP2 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.

```
ZIP1.8, doubleword
Vd A7 A6 A5 A4 A3 A2 A1 A0
```

```
ZIP2.8, doubleword
Vn 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.

**Advanced SIMD**

ZIP1 `<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;
integer part = `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>

`<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 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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
ZIP2

Zip vectors (secondary). 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 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 |
Vd | 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.

### 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;

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>.

**FACTL:** 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).
FEEXPA: Floating-point exponential accelerator.
FLOGB: 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 (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.
FMINNNM: 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).
FMLLS (indexed): Floating-point fused multiply-subtract by indexed elements (Zda = Zda + -Zn * Zm[indexed]).
FMLLS (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].
FNMA: Floating-point negated fused multiply-add vectors (predicated), writing addend [Zda = -Zda + -Zn * Zm].
FNMLA: Floating-point negated fused multiply-add 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\(<r>\)**: Floating-point round to integral value (predicated).

**FRSORTE**: Floating-point reciprocal square root estimate (unpredicated).

**FRSORTS**: Floating-point reciprocal square root step (unpredicated).

**FSCALF**: 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.

**FTSSFL**: 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).
LD1RQB (scalar plus immediate): Contiguous load and replicate sixteen bytes (immediate index).
LD1RQB (scalar plus scalar): Contiguous load and replicate sixteen bytes (scalar index).
LD1RQD (scalar plus immediate): Contiguous load and replicate two doublewords (immediate index).
LD1RQD (scalar plus scalar): Contiguous load and replicate two doublewords (scalar index).
LD1RQH (scalar plus immediate): Contiguous load and replicate eight halfwords (immediate index).
LD1RQH (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 (scalar plus vector)**: Gather load unsigned words to vector (vector index).

**LD1W (vector plus immediate)**: Gather load unsigned words to vector (immediate 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).

**LD2B (vector 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).

**LD2D (vector 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).

**LD2H (vector 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).

**LD3B (vector 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).

**LD3D (vector 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).

**LD3H (vector 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).

**LD4B (vector 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).

**LD4D (vector 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).

**LD4H (vector 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 multipicand [Zdn = Za + Zdn * 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 * Zm].

MLS (indexed): Multiply-subtract from accumulator (indexed).

MLS (vectors): Multiply-subtract vectors (predicated), writing addend [Zda = Zda - Zn * Zm].


MOV: Move predicate (unpredicated): an alias of ORR, ORRS (predicates).

MOV: Move predicates (merging): an alias of SEL (predicates).

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).

**MOVPRFX (predicated):** Move prefix (predicated).

**MOVPRFX (unpredicated):** Move prefix (unpredicated).

**MOVS:** Move predicates (zeroing), setting the condition flags: an alias of AND, ANDS (predicates).

**MOVS:** Move predicate (unpredicated), setting the condition flags: an alias of ORR, ORRS (predicates).

**MSB:** Multiply-subtract vectors (predicated), writing multiplicand $[Zdn = Za - Zdn \times Zm]$.

**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, NORS:** 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.
RADDHNB: Rounding add narrow high part (bottom).
RADDHNT: Rounding add narrow high part (top).
RAXI: Bitwise rotate left by 1 and exclusive OR.
RBIT: Reverse bits (predicated).
RDFFR (unpredicated): Read the first-fault register.
RDFFR, RDFFRS (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).
RSHRNH: Rounding shift right narrow by immediate (bottom).
RSHRNT: Rounding shift right narrow by immediate (top).
RSUBHNB: Rounding subtract narrow high part (bottom).
RSUBHNT: 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).
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SABD</td>
<td>Signed absolute difference (predicated).</td>
</tr>
<tr>
<td>SABDLB</td>
<td>Signed absolute difference long (bottom).</td>
</tr>
<tr>
<td>SABDLT</td>
<td>Signed absolute difference long (top).</td>
</tr>
<tr>
<td>SADALP</td>
<td>Signed add and accumulate long pairwise.</td>
</tr>
<tr>
<td>SADDLB</td>
<td>Signed add long (bottom).</td>
</tr>
<tr>
<td>SADDLBT</td>
<td>Signed add long (bottom + top).</td>
</tr>
<tr>
<td>SADLT</td>
<td>Signed add long (top).</td>
</tr>
<tr>
<td>SADDY</td>
<td>Signed add reduction to scalar.</td>
</tr>
<tr>
<td>SADDWB</td>
<td>Signed add wide (bottom).</td>
</tr>
<tr>
<td>SADDWT</td>
<td>Signed add wide (top).</td>
</tr>
<tr>
<td>SBCLB</td>
<td>Subtract with carry long (bottom).</td>
</tr>
<tr>
<td>SBCLT</td>
<td>Subtract with carry long (top).</td>
</tr>
<tr>
<td>SCVTF</td>
<td>Signed integer convert to floating-point (predicated).</td>
</tr>
<tr>
<td>SDIV</td>
<td>Signed divide (predicated).</td>
</tr>
<tr>
<td>SDIVR</td>
<td>Signed reversed divide (predicated).</td>
</tr>
<tr>
<td>SDOT (indexed)</td>
<td>Signed integer indexed dot product.</td>
</tr>
<tr>
<td>SDOT (vectors)</td>
<td>Signed integer dot product.</td>
</tr>
<tr>
<td>SEL (predicates)</td>
<td>Conditionally select elements from two predicates.</td>
</tr>
<tr>
<td>SEL (vectors)</td>
<td>Conditionally select elements from two vectors.</td>
</tr>
<tr>
<td>SETFFR</td>
<td>Initialise the first-fault register to all true.</td>
</tr>
<tr>
<td>SHADD</td>
<td>Signed halving addition.</td>
</tr>
<tr>
<td>SHRNB</td>
<td>Shift right narrow by immediate (bottom).</td>
</tr>
<tr>
<td>SHRNT</td>
<td>Shift right narrow by immediate (top).</td>
</tr>
<tr>
<td>SHSUB</td>
<td>Signed halving subtract.</td>
</tr>
<tr>
<td>SHSUBR</td>
<td>Signed halving subtract reversed vectors.</td>
</tr>
<tr>
<td>SLI</td>
<td>Shift left and insert (immediate).</td>
</tr>
<tr>
<td>SM4E</td>
<td>SM4 encryption and decryption.</td>
</tr>
<tr>
<td>SM4EKEY</td>
<td>SM4 key updates.</td>
</tr>
<tr>
<td>SMAX (immediate)</td>
<td>Signed maximum with immediate (unpredicated).</td>
</tr>
<tr>
<td>SMAX (vectors)</td>
<td>Signed maximum vectors (predicated).</td>
</tr>
<tr>
<td>SMAXP</td>
<td>Signed maximum pairwise.</td>
</tr>
<tr>
<td>SMAXV</td>
<td>Signed maximum reduction to scalar.</td>
</tr>
<tr>
<td>SMIN (immediate)</td>
<td>Signed minimum with immediate (unpredicated).</td>
</tr>
<tr>
<td>SMIN (vectors)</td>
<td>Signed minimum vectors (predicated).</td>
</tr>
<tr>
<td>SMINP</td>
<td>Signed minimum pairwise.</td>
</tr>
<tr>
<td>SMINY</td>
<td>Signed minimum reduction to scalar.</td>
</tr>
</tbody>
</table>
**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.

**SOADD (immediate):** Signed saturating add immediate (unpredicated).

**SOADD (vectors, predicated):** Signed saturating addition (predicated).

**SOADD (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.

**SQDECW (scalar):** Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.

**SQDECW (vector):** Signed saturating decrement vector by multiple of 32-bit predicate constraint element count.

**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).

**SODMLSLB (indexed):** Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed).
**SQDMLSLB (vectors)**: Signed saturating doubling multiply-subtract long from accumulator (bottom).

**SQDMLSLBT**: Signed saturating doubling multiply-subtract long from accumulator (bottom × top).

**SQDMLSLT (indexed)**: Signed saturating doubling multiply-subtract long from accumulator (top, indexed).

**SQDMLSLT (vectors)**: Signed saturating doubling multiply-subtract long from accumulator (top).

**SQDMULH (indexed)**: Signed saturating doubling multiply high (indexed).

**SQDMULH (vectors)**: Signed saturating doubling multiply high (unpredicated).

**SQDMLULB (indexed)**: Signed saturating doubling multiply long (bottom, indexed).

**SQDMLULB (vectors)**: Signed saturating doubling multiply long (bottom).

**SQDMLULT (indexed)**: Signed saturating doubling multiply long (top, indexed).

**SQDMLULT (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).

**SQRDMMLSH (indexed)**: Signed saturating rounding doubling multiply-subtract high from accumulator (indexed).

**SQRDMMLSH (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).

**SQRSHRNP**: Signed saturating rounding shift right narrow by immediate (bottom).

**SQRSHRNT**: Signed saturating rounding shift right narrow by immediate (top).

**SQRSHRUNB**: 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).
SOTXNB: Signed saturating extract narrow (bottom).
SOTXTN: Signed saturating extract narrow (top).
SOTXUNB: Signed saturating unsigned extract narrow (bottom).
SOTXTUNT: 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 (immediate index).
ST1H (vector plus scalar): Scatter store halfwords from a vector (scalar index).
ST1W (scalar plus immediate): Contiguous store words from vector (immediate index).
ST1W (scalar plus scalar): Contiguous store words from vector (scalar index).
ST1W (vector plus immediate): Scatter store words from a vector (immediate index).
ST1W (vector plus scalar): Scatter store words from a vector (scalar 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).
UABDLB: 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.
UMINV: 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).
UQDECB: Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count.
UQDECD (scalar): Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count.
UQDECD (vector): Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count.
UQDECH (scalar): Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count.
UQDECH (vector): Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count.
UQDECP (scalar): Unsigned saturating decrement scalar by count of true predicate elements.
UQDECP (vector): Unsigned saturating decrement vector by count of true predicate elements.
UQDECW (scalar): Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count.
UQDECW (vector): Unsigned 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).
UQSHRNB (bottom): Unsigned saturating shift right narrow by immediate.
UQSHRNT (top): Unsigned saturating shift right narrow by immediate.
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).
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 (bottom): Unsigned saturating extract narrow.
UQXTNT (top): Unsigned saturating extract narrow.
URECPE: Unsigned reciprocal estimate (predicated).
URHADD: Unsigned rounding halving addition.
URSHL (vector): 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.

WHILERW: While free of read-after-write conflicts.

WHILEWR: While free of write-after-read/write conflicts.

WRFFR: 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

SVE

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(Operand[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

```
   31  30  29  28  27  26  25  24  23  22  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  0  0  1  0  1  0  0  1  1  0  1  0  1  1  0
```

SVE2

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":

```
sz  <T>
 0  S
 1  D
```

<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>;<l>
    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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

SVE2

ADCLT \(<\text{Zda}>.<T>, <\text{Zn}>.<T>, <\text{Zm}>.<T>\)

if !\(\text{HaveSVE2}()\) then UNDEFINED;
integer esize = 32 \(<\text{UInt}(sz)\);
integer n = \(\text{UInt}(\text{Zn})\);
integer m = \(\text{UInt}(\text{Zm})\);
integer da = \(\text{UInt}(\text{Zda})\);

Assembler Symbols

\(<\text{Zda}>\) 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>

\(<\text{Zn}>\) 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 pairs = \(\text{VL} \div (\text{esize} \times 2)\);
bits(\(\text{VL}\)) operand = \(\text{Z[n]}\);
bits(\(\text{VL}\)) carries = \(\text{Z[m]}\);
bits(\(\text{VL}\)) result = \(\text{Z[da]}\);

for p = 0 to pairs-1
    bits(\(\text{esize}\)) element1 = \(\text{Elem}[\text{result}, 2*p + 0, \text{esize}]\);
    bits(\(\text{esize}\)) element2 = \(\text{Elem}[\text{operand}, 2*p + 1, \text{esize}]\);
    bit carry_in = \(\text{Elem}[\text{carries}, 2*p + 1, \text{esize}]<0>\);
    \((\text{res}, \text{nzcv}) = \text{AddWithCarry}(\text{element1}, \text{element2}, \text{carry_in});\)
    \(\text{carry_out} = \text{nzcv}<1>\);
    \(\text{Elem}[\text{result}, 2*p + 0, \text{esize}] = \text{res};\)
    \(\text{Elem}[\text{result}, 2*p + 1, \text{esize}] = \text{ZeroExtend}(\text{carry_out});\)
\(\text{Z[da]} = \text{result};\)

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.
This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 0 1 0 0 | size      | 0 0 0 0 0 0 0 0 0 0 0 | Pg        | Zm        | Zdn        |

SVE

ADD <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 + 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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 “#<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”.

```

<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>size</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>size</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>sh</td>
<td>imm8</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>
```

SVE

ADD <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;

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”:
  - size | <T>
  - 00 | B
  - 01 | H
  - 10 | S
  - 11 | D
- **<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”:
  - sh | <shift>
  - 0 | LSL #0
  - 1 | LSL #8

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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 0 0 1 0 0 | size | Zm | 0 0 0 0 0 0 | Zn | Zd |

SVE

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 align="center">size</th>
<th align="center">&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td align="center">00</td>
<td align="center">B</td>
</tr>
<tr>
<td align="center">01</td>
<td align="center">H</td>
</tr>
<tr>
<td align="center">10</td>
<td align="center">S</td>
</tr>
<tr>
<td align="center">11</td>
<td align="center">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.
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  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  |

**SVE2**

ADDHNB <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;
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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 1  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| size| Zm | 0  | 1  | 1  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Zn  | Zd |

SVE2

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

\[ \text{CheckSVEEnabled}(); \]
integer elements = \( \text{VL} \div \text{esize} \);
bits(\text{VL}) operand1 = \( Z[n] \);
bits(\text{VL}) operand2 = \( Z[m] \);
bits(\text{VL}) result = \( Z[d] \);
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\ast e + 1, halfesize] = res<halfesize-1:0>;
\]
\[ Z[d] = \text{result}; \]

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.
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  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  |

**SVE2**

ADDP <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'

        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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | Rn | 0  | 1  | 0  | 1  | 0  | imm6 | Rd |
```

**SVE**

`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.

<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 0 1</td>
</tr>
</tbody>
</table>

SVE

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | | sz | 1  | Zm | 1  | 0  | 1  | 0  | msz | Zn | | Zd |

### Unpacked 32-bit signed offsets

ADR \(<Zd>\).\(<T>\), \([<Zn>].\(<T>\), \(<Zm>].\(<T>\>{\{, \<mod> \<amount>\}}\)

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 unsigned offsets

ADR \(<Zd>.D, \[<Zn>.D, \(<Zm>.D, SXTW{ \<amount>\}}\)

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

ADR \(<Zd>\).D, \[<Zn>\].D, \(<Zm>\).D, SXTW\{ \<amount>\}\)

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

ADR <Zd>.D, [<Zn>.D, <Zm>.D, UXTW{ <amount>}]

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”:

<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 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

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  0  1  0  1  0  0  0  1  0  1  1  1  0  0  1  Zm | Zdn

SVE2


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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 1 1 1 0 0 0 1 0 1 1 0 0 0 | Zm   | Zdn |

SVE2


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 \texttt{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.

\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 0 0 1 0 1 0 0 0 0 0 1 1 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 Zdn
\end{verbatim}

SVE2


if !\texttt{HaveSVE2AES()} then UNDEFINED;
integer dn = \texttt{UInt}(Zdn);

Assembler Symbols

<Zdn> is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

\begin{verbatim}
\texttt{CheckSVEEnabled()};
integer segments = \texttt{VL} \texttt{DIV} 128;
bits(\texttt{VL}) operand = \texttt{Z}[dn];
bits(\texttt{VL}) result;

for s = 0 to segments-1
   \texttt{Elem}[result, s, 128] = AESInvMixColumns(\texttt{Elem}[operand, s, 128]);

\texttt{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.
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 \texttt{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.

\texttt{ID\_AA64ZFR0\_EL1.AES} indicates whether this instruction is implemented.

\begin{verbatim}
0 1 0 0 1 0 1 0 | 0 1 0 0 0 0 0 1 | 1 1 0 0 0 | 0 0 0 0 0 | 0 0 0 0 0 | Zdn
\end{verbatim}

SVE2


if \texttt{!HaveSVE2AES()} then UNDEFINED;
integer \texttt{dn} = \texttt{UInt}(Zdn);

Assembler Symbols

\texttt{<Zdn>} is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

CheckSVEEnabled();
integer segments = \texttt{VL} \texttt{DIV} 128;
bits(\texttt{VL}) operand = \texttt{Z[dn]};
bits(\texttt{VL}) result;
for \texttt{s} = 0 to segments-1
\quad \texttt{Elem[result, s, 128]} = AESMixColumns(\texttt{Elem[operand, s, 128]});
\texttt{Z[dn]} = result;

Operational information

If \texttt{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}
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 MOVS, and MOV.

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 1 0 1 0 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 |

\[ \text{Pm} \quad \text{Pg} \quad \text{Pn} \quad \text{Pd} \]

Not setting the condition flags

\[
\text{AND} \ <\text{Pd}>.B, \ <\text{Pg}>/Z, \ <\text{Pn}>.B, \ <\text{Pm}>.B \]

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 |
|----------------------------------|----------------------------------|----------------------------------|----------------------------------|
| 0 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 |

\[ \text{Pm} \quad \text{Pg} \quad \text{Pn} \quad \text{Pd} \]

Setting the condition flags

\[
\text{ANDS} \ <\text{Pd}>.B, \ <\text{Pg}>/Z, \ <\text{Pn}>.B, \ <\text{Pm}>.B \]

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

\(<\text{Pd}>\)
Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\(<\text{Pg}>\)
Is the name of the governing scalable predicate register, encoded in the "Pg" field.

\(<\text{Pn}>\)
Is the name of the first source scalable predicate register, encoded in the "Pn" field.

\(<\text{Pm}>\)
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>MOVS</td>
<td>$S == '1'$ &amp;&amp; $\text{Pn} == \text{Pm}$</td>
</tr>
</tbody>
</table>
Alias | Is preferred when
---|---
MOV | S == '0' & Pn == Pm

**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 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.
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.

| 31 30 29 28 27 26 25 24 23 22 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 1 0 | 0 0 0 | Pg | Zm | Zdn |

SVE

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":

<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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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).

31 30 29 28 27 26 25 24 23 22 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 0 0 0 imm13
Zdn

SVE

AND <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>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

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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | Zm | 0 | 0 | 1 | 1 | 0 | 0 | Zn | Zd |

SVE

\[
\text{AND } <\text{Zd}>.D, <\text{Zn}>.D, <\text{Zm}>.D
\]

if \( \text{!HaveSVE}() \) then UNDEFINED;
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.
\(<\text{Zn}>\) 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

\[
\text{CheckSVEEnabled}();
\]
\[
\text{bits(VL)} \text{ operand1} = Z[n];
\]
\[
\text{bits(VL)} \text{ operand2} = Z[m];
\]
\[
Z[d] = \text{operand1 AND operand2};
\]

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.

<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>

SVE

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":

<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 = 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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**SVE**

ASR <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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
AR (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.

```
0 0 0 0 0 1 0 0 | size | 0 1 1 0 0 1 0 0 | Pg | Zm | Zdn
```

SVE

AR <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":

```
size | <T>
-----|-----
0 0  | B
0 1  | H
1 0  | S
1 1  | RESERVED
```

<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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and destination element size as 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  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 0 1 0 0 | tzs | 0 1 1 0 0 0 0 0 | Zn | Zd

SVE

ASR <Zd>.<T>, <Zn>.<T>, #<const>

if !HaveSVE() then UNDEFINED;
bits(4) tsize = tzs:tsl;
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 “tsz:tsl”:

<table>
<thead>
<tr>
<th>tzs</th>
<th>tsl</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 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) 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | Zm | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Zn | Zd |

SVE

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  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE

ASRD <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>tshr</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 "tszh:imm3".

Operation

CheckSVEEnabled();
integer elements = VL DIV esize;
bits(PG) mask = P[g];
bits(VL) operandl = Z[dn];
bits(VL) result;
for e = 0 to elements-1
  integer elementl = SInt(Elem[operandl, e, esize]);
  if Elem[mask, e, esize] == '1' then
    if elementl < 0 then
      elementl = elementl + ((1 << shift) - 1);
    Elem[result, e, esize] = (elementl >> shift)<esize-1:0>;
  else
    Elem[result, e, esize] = Elem[operandl, 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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 | 0 1 0 1 0 0 1 0 0 | Pg | Zm | Zdn |

SVE

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 ElemP[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

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

Zm  0 0 1 1 1 0 1 0 0 0 0 0 0 0 0
Zk  0 0 1 1 1 0 1 0 0 0 0 0 0 0 0
Zdn 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0

SVE2


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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX 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.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & size & 0 & Zm & 1 & 0 & 1 & 1 & 0 & 1 & Zn & Zd
\end{array}
\]

SVE2

BDEP \text{<Zd>.<T>, <Zn>.<T>, <Zm>.<T>}

if !\text{HaveSVE2BitPerm}() 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”:
\[
\begin{array}{c|c}
size & \text{<T>} \\
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\end{array}
\]
\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} / esize;
bits(\text{VL}) data = Z[n];
bits(\text{VL}) mask = Z[m];
bits(\text{VL}) result;
for e = 0 to elements - 1
\text{Elem[result, e, esize]} = \text{BitDeposit(Elem[data, e, esize], Elem[mask, e, esize]);}
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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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 | 1  | 0  | 1  | 1  | 0  | 0  | Zn | 1  | Zd |

SVE2

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.

SVE

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 Elem[mask, e, 32] == '1' then
    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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

![Register encoding](image)

**SVE**

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.

ID AA64ZFR0_EL1.BF16

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Zm | 1  | 0  | 0  | 0  | 0  | 0  | 0  | Zn | 0  | 0  | 0  | 0  | 0  | Zda |

SVE


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(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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

CheckSVEEnabled();
integer elements = \texttt{VL} \text{ DIV } 32;
integer eltspersegment = 128 \text{ DIV } 32;
bits(\texttt{VL}) operand1 = \texttt{Z}[n];
bits(\texttt{VL}) operand2 = \texttt{Z}[m];
bits(\texttt{VL}) operand3 = \texttt{Z}[da];
bits(\texttt{VL}) result;

for e = 0 to elements - 1
  integer segmentbase = e - e \text{ MOD } \text{ eltspersegment};
  integer s = segmentbase + index;
  bits(16) elt1_a = \texttt{Elem}[operand1, 2 * e + 0, 16];
  bits(16) elt1_b = \texttt{Elem}[operand1, 2 * e + 1, 16];
  bits(16) elt2_a = \texttt{Elem}[operand2, 2 * s + 0, 16];
  bits(16) elt2_b = \texttt{Elem}[operand2, 2 * s + 1, 16];

  bits(32) sum = \texttt{BFAdd}(\texttt{BFMul}(elt1_a, elt2_a), \texttt{BFMul}(elt1_b, elt2_b));
  \texttt{Elem}[result, e, 32] = \texttt{BFAdd}(%texttt{Elem}[operand3, e, 32], sum);

\texttt{Z}[da] = result;

Operational information

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- 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.

The \texttt{MOVPRFX} instructions that can be used with this instruction are as follows:

- An unpredicated \texttt{MOVPRFX} 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.

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 + 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] = FP MullAdd(element3, element1, element2, FPCR);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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   | 0   | 1   | 1   | 1   | i3h | Zm  | 0   | 1   | 0   | i3l | 0   | Zn  | 0   | Zda |

SVE


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] = FPMulAdd(element3, element1, element2, FPCR);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The **MOVPRFX** instructions that can be used with this instruction are as follows:

- An unpredicated **MOVPRFX** instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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_A64ZFR0_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 | Zm | 1 0 0 0 0 1 | Zn | Zda

SVE


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] = FPMulAdd(element3, element1, element2, FPCR);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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 | 1 | Zn | | | | | | | | | | | |

SVE


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] = FPMulAdd(element3, element1, element2, FPCR);
Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
BFMMLA

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.
* 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

SVE2

BGRP <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);

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>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, 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

<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>Pm</td>
<td>0</td>
<td>1</td>
<td>Pg</td>
<td>0</td>
<td>Pn</td>
<td>1</td>
<td>Pd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

### Setting the condition flags

Not setting the condition flags


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

Setting the condition flags


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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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>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>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></td>
<td></td>
</tr>
</tbody>
</table>

SVE

BIC \(<Zdn>\).\(<T>\), \(<Pg>/M\), \(<Zdn>\).\(<T>\), \(<Zm>\).\(<T>\)

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});

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

\text{CheckSVEEnabled}();
integer elements = \text{VL} \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
  bits(esize) element1 = \text{Elem}[\text{operand1}, e, \text{esize}];
  bits(esize) element2 = \text{Elem}[\text{operand2}, e, \text{esize}];
  if \text{ElemP}[mask, e, \text{esize}] == '1' then
    \text{Elem}[result, e, \text{esize}] = element1 AND (NOT element2);
  else
    \text{Elem}[result, e, \text{esize}] = \text{Elem}[\text{operand1}, e, \text{esize}];
\text{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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

```
0 0 0 0 1 0 0 1 1 1  Zm
0 0 1 1 0 0  Zn
  Zd
```

**SVE**

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

```
0 0 1 0 0 1 0 1 | 0 0 1 0 0 0 0 0 1 | Pg | 0 | Pn | M | Pd
```

### Setting the condition flags

```
0 0 1 0 0 1 0 1 | 0 1 0 1 0 0 0 0 1 | Pg | 0 | Pn | 0 | 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 source scalable predicate register, encoded in the "Pn" field.
- `<ZM>` Is the predication qualifier, encoded in "M":

| M | <ZM>
|---|---
| 0 | Z
| 1 | M

- BRKA `<Pd>`.B, `<Pg>`/<ZM>, `<Pn>`.B
- BRKAS `<Pd>`.B, `<Pg>`/<Z>, `<Pn>`.B
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;
```

Operational information

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.
**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 1 0 0 0 0 0 1 | Pg | 0 | Pn | M | Pd |
```

**BRKB <Pd>.B, <Pg>/<ZM>, <Pn>.B**

```java
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 | Pd |
```

**BRKBS <Pd>.B, <Pg>/Z, <Pn>.B**

```java
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;
    endif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';
    endif

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.
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 0 1 0 1 0 0 1 1 0 0 0 0 1 | Pg 0 | Pn 0 | Pdm |

Not setting the condition flags


if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = 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 0 1 0 1 0 1 0 0 0 1 | Pg 0 | Pn 0 | Pdm |

Setting the condition flags


if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = 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

```c
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 |
|   |   |   |   |   | Pm | 1 | Pg |   | Pn | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Not setting the condition flags


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 |
|   |   |   |   |   | Pm | 1 | Pg |   | Pn | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Setting the condition flags


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

| 31 30 29 28 27 26 25 24 23 22 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 | Pm | 1 | 1 | Pg | 0 | Pn | 1 | Pd |

Not setting the condition flags


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 0 1 | 0 | 1 | 0 | 0 | Pm | 1 | 1 | Pg | 0 | Pn | 1 | Pd |

Setting the condition flags


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;
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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 30 29 28 27 26 25 24 23 22 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 1 1 1 1                 | Zm                            |
|                               |                               | Zk                            |
|                               |                               | Zdn                           |

SVE2


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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

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

Zm 0 0 1 1 1 1
Zk 0 0 1 1 1 1
Zdn 0 0 1 1 1 1

SVE2


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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX instruction.
**BSL**

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.

<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 1</td>
</tr>
</tbody>
</table>

**SVE2**


```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();
bits(VL) operand1 = Z[dn];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[k];
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE2

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

CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
bv[VL] operand1 = Z[dn];
bv[VL] operand2 = Z[m];
bv[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;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
0 1 0 0 0 1 0 0 | size 0 | Zm 0 0 0 1 | rot Zn | Zda
```

**SVE2**

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>);
```

**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”:

CDOT (vectors)
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

If PSTATE.DIT is 1:

• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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  | 1  | 0  | 1  | 2  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

\[ \text{CDOT} <\text{Zda}>, \text{S}, <\text{Zn}>, \text{B}, <\text{Zm}>, \text{B}[\text{<imm>}], <\text{const}> \]
```

if !\text{HaveSVE2}() then UNDEFINED;
integer esize = 32;
integer index = \text{UInt}(i2);
integer n = \text{UInt}(\text{Zn});
integer m = \text{UInt}(\text{Zm});
integer da = \text{UInt}(\text{Zda});
integer sel_a = \text{UInt}(\text{rot<0>});
integer sel_b = \text{UInt}(\text{NOT(\text{rot<0>})});
boolean sub_i = (\text{rot<0>} == \text{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  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
```

CDOT (indexed)
64-bit


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.

<constants> 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

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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 0 1 0 1 size 1 1 0 0 0 0 1 0 1 Pg Zm Rdn

SVE

CLASTA <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 = FALSE;

Assembler Symbols

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.

size <T>
00 B
01 H
10 S
11 D

Operation

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;
Operational information

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.
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  | 1  | 0  | 1  | 0  | 1  | 0  |
| 1   | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  |
| Pg  | Zm | Vdn |

SVE

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;
Operational information

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.
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 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 |

SVE

CLASTA <Zdn>.<T>, <Pg>, <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 isBefore = 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;
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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 30 29 28 27 26 25 24 23 22 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 1 0 1 | Pg | Zm | Rdn |

SVE

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”:

<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

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;
Operational information

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.
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  | 0  | 1  | 1  | 1  | 0  | 0  | Pg | Zm | Vdn |

SVE

CLASTB <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 = 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

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;
Operational information

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.
**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.

```
| 31 30 29 28 27 26 25 24 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               |
```

**SVE**

CLASTB $<Zdn>$.<T>, $<Pg>$, $<Zdn>$.<T>, $<Zm>$.<T>

```java
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**

```java
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**

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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & P_{g} & Z_{n} & Z_{d}
\end{array}
\]

SVE

\[
\text{CLS}<Z_d>,<T>,<P_g>/M,<Z_n>,<T>
\]

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(P_g);
integer n = UInt(Z_n);
integer d = UInt(Z_d);

Assembler Symbols

- \(<Z_d>\) 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>

- \(<P_g>\) is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \(<Z_n>\) is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

\[
\text{CheckSVEEnabled}();
\]

integer elements = \(\text{VL} / \text{esize}\);
bits(PL) mask = \(P[g]\);
bits(VL) operand = \(Z[n]\);
bits(VL) result = \(Z[d]\);

for \(e = 0\) to \(\text{elements-1}\)
\[
\begin{align*}
\text{bits}(\text{esize}) \text{ element} &= \text{Elem}[^{\text{operand}}, e, \text{esize}]; \\
\text{if} \text{Elem}[\text{mask}, e, \text{esize}] == '1' \text{ then} \\
& \quad \text{Elem}[^{\text{result}}, e, \text{esize}] = \text{CountLeadingSignBits}[^{\text{element}}]<\text{esize-1:0}>;
\end{align*}
\]

\(Z[d] = \text{result};\)

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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

\[
\begin{array}{cccccccccccccccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 0 & | & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & | & Pg & | & Zn & | & Zd \\
\end{array}
\]

### SVE

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><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
    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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

### SVE2

CMLA `<Zda>,<T>, <Zn>,<T>, <Zm>,<T>, <const>`

```assembly
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

```
OperationCheckSVEEnabled();
integer pairs = \texttt{VL} \div (2 * \texttt{esize});
bits(\texttt{VL}) operand1 = \texttt{Z}[n];
bits(\texttt{VL}) operand2 = \texttt{Z}[m];
bits(\texttt{VL}) operand3 = \texttt{Z}[da];
bits(\texttt{VL}) result;

\textbf{for} p = 0 \textbf{to} pairs-1
\begin{itemize}
  \item integer elt1_a = \texttt{SInt}(\texttt{Elem}[operand1, 2 * p + sel_a, \texttt{esize}]);
  \item integer elt2_a = \texttt{SInt}(\texttt{Elem}[operand2, 2 * p + sel_a, \texttt{esize}]);
  \item integer elt2_b = \texttt{SInt}(\texttt{Elem}[operand2, 2 * p + sel_b, \texttt{esize}]);
  \item bits(\texttt{esize}) elt3_r = \texttt{Elem}[operand3, 2 * p + 0, \texttt{esize}];
  \item bits(\texttt{esize}) elt3_i = \texttt{Elem}[operand3, 2 * p + 1, \texttt{esize}];
  \item integer product_r = elt1_a * elt2_a;
  \item integer product_i = elt1_a * elt2_b;
  \item if sub_r then
    \item \texttt{Elem}[result, 2 * p + 0, \texttt{esize}] = elt3_r - product_r;
  \item else
    \item \texttt{Elem}[result, 2 * p + 0, \texttt{esize}] = elt3_r + product_r;
  \item if sub_i then
    \item \texttt{Elem}[result, 2 * p + 1, \texttt{esize}] = elt3_i - product_i;
  \item else
    \item \texttt{Elem}[result, 2 * p + 1, \texttt{esize}] = elt3_i + product_i;
\end{itemize}
Z[da] = 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.

This instruction might be immediately preceded in program order by a \texttt{MOVPRFX} instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is \texttt{UNPREDICTABLE}:
- 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.

The \texttt{MOVPRFX} instructions that can be used with this instruction are as follows:
- An unpredicated \texttt{MOVPRFX} 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

| 31 30 29 28 27 26 25 24 23 22 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 1 1 0 | rot | Zn | Zda |

16-bit


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 1 1 | i1 | Zm | 0 1 1 0 | rot | Zn | Zda |

32-bit

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.
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

\[\text{CheckSVEEnabled}();\]
integer pairs = \(VL \div (2 \times \text{esize})\);
integer pairspersegment = 128 \div (2 \times \text{esize}) ;
bits(VL) operand1 = Z[n] ;
bits(VL) operand2 = Z[m] ;
bits(VL) operand3 = Z[da] ;
bits(VL) result ;
for \(p = 0\) to \(\text{pairs-1}\)
  integer segmentbase = \(p - p \mod \text{pairspersegment}\);
  integer s = segmentbase + index ;
  integer elt1_a = \(\text{SInt} (\text{Elem}(\text{operand1}, 2 \times p + \text{sel} \_\text{a} , \text{esize}))\) ;
  integer elt2_a = \(\text{SInt} (\text{Elem}(\text{operand2}, 2 \times s + \text{sel} \_\text{a} , \text{esize}))\) ;
  integer elt2_b = \(\text{SInt} (\text{Elem}(\text{operand2}, 2 \times s + \text{sel} \_\text{b} , \text{esize}))\) ;
  bits(\text{esize}) elt3_r = \(\text{Elem}(\text{operand3}, 2 \times s + \text{sel} \_\text{a} , \text{esize})\) ;
  bits(\text{esize}) elt3_i = \(\text{Elem}(\text{operand3}, 2 \times s + \text{sel} \_\text{b} , \text{esize})\) ;
  integer product_r = elt1_a * elt2_a ;
  integer product_i = elt1_a * elt2_b ;
  if \text{sub}_r\ then
    \(\text{Elem}(\text{result}, 2 \times p + 0 , \text{esize}) = \text{elt3} \_\text{r} - \text{product} \_\text{r} ;\)
  else
    \(\text{Elem}(\text{result}, 2 \times p + 0 , \text{esize}) = \text{elt3} \_\text{r} + \text{product} \_\text{r} ;\)
  if \text{sub}_i\ then
    \(\text{Elem}(\text{result}, 2 \times p + 1 , \text{esize}) = \text{elt3} \_\text{i} - \text{product} \_\text{i} ;\)
  else
    \(\text{Elem}(\text{result}, 2 \times p + 1 , \text{esize}) = \text{elt3} \_\text{i} + \text{product} \_\text{i} ;\)
\(\text{Z}[\text{da}] = \text{result} ;\)

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.

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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**

```
0 0 1 0 0 1 0 1 | size 0 | imm5 1 0 0 | Pg | Zn | 0 | Pd
```

**Greater than**

```
0 0 1 0 0 1 0 1 | size 0 | imm5 0 0 0 | Pg | Zn 1 | Pd
```

**Greater than or equal**

```
0 1 0 0 1 0 1 | size 0 | imm5 0 0 0 | Pg | Zn 0 | Pd
```

```plaintext
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 or equal

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 = UInt(imm5);
boolean unsigned = FALSE;

Higher

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

CMPSH <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

CMP<cc> (immediate)
Less than

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

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(imm7);
boolean unsigned = FALSE;

Lower

CMPLO <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

CMPL (immediate)
Lower or same

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

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.
<T> Is the 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.

<imm> 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

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;
    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.

Copyright © 2010-2019 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

```
31 30 29 28 27 26 25 24 23 22 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 0 1 Pg Zn 0 Pd
```

Equal

```
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

```
31 30 29 28 27 26 25 24 23 22 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 0 Pg Zn 1 Pd
```

Greater than

```
CMPGT <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

```
31 30 29 28 27 26 25 24 23 22 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 0 Pg Zn 0 Pd
```
Greater than or equal

CMPGE <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 = FALSE;

Higher

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

CMPHS <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

CMP<cc> (wide elements)
Less than

\[
\text{CMPLT } \langle \text{Pd} \rangle.\langle T \rangle, \langle \text{Pg} \rangle/\langle Z \rangle, \langle \text{Zn} \rangle.\langle T \rangle, \langle \text{Zm} \rangle.\langle D \rangle
\]

if !\texttt{HaveSVE}() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer g = \texttt{UInt}(Pg);
integer n = \texttt{UInt}(Zn);
integer m = \texttt{UInt}(Zm);
integer d = \texttt{UInt}(Pd);
\texttt{SVEcmp} op = \texttt{Cmp_LT};
boolean unsigned = FALSE;

Less than or equal

\[
\text{CMPLE } \langle \text{Pd} \rangle.\langle T \rangle, \langle \text{Pg} \rangle/\langle Z \rangle, \langle \text{Zn} \rangle.\langle T \rangle, \langle \text{Zm} \rangle.\langle D \rangle
\]

if !\texttt{HaveSVE}() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer g = \texttt{UInt}(Pg);
integer n = \texttt{UInt}(Zn);
integer m = \texttt{UInt}(Zm);
integer d = \texttt{UInt}(Pd);
\texttt{SVEcmp} op = \texttt{Cmp_LE};
boolean unsigned = FALSE;

Lower

\[
\text{CMPL} \langle \text{Pd} \rangle.\langle T \rangle, \langle \text{Pg} \rangle/\langle Z \rangle, \langle \text{Zn} \rangle.\langle T \rangle, \langle \text{Zm} \rangle.\langle D \rangle
\]

if !\texttt{HaveSVE}() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer g = \texttt{UInt}(Pg);
integer n = \texttt{UInt}(Zn);
integer m = \texttt{UInt}(Zm);
integer d = \texttt{UInt}(Pd);
\texttt{SVEcmp} op = \texttt{Cmp_LT};
boolean unsigned = TRUE;

Lower or same

\[
\text{CMPO } \langle \text{Pd} \rangle.\langle T \rangle, \langle \text{Pg} \rangle/\langle Z \rangle, \langle \text{Zn} \rangle.\langle T \rangle, \langle \text{Zm} \rangle.\langle D \rangle
\]

if !\texttt{HaveSVE}() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << \texttt{UInt}(size);
integer g = \texttt{UInt}(Pg);
integer n = \texttt{UInt}(Zn);
integer m = \texttt{UInt}(Zm);
integer d = \texttt{UInt}(Pd);
\texttt{SVEcmp} op = \texttt{Cmp_LT};
boolean unsigned = TRUE;
Lower or same

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

CMPNE <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_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

```c
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;

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

\[
\begin{array}{cccccccccccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & Zm & 1 & 0 & 1 & Pg & Zn & 0 & Pd
\end{array}
\]

Equal

\[
\text{CMPEQ } <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
\]

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

\[
\begin{array}{cccccccccccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & Zm & 1 & 0 & 0 & Pg & Zn & 1 & Pd
\end{array}
\]

Greater than

\[
\text{CMPGT } <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>
\]

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

\[
\begin{array}{cccccccccccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \text{size} & 0 & Zm & 1 & 0 & 0 & Pg & Zn & 0 & Pd
\end{array}
\]
Greater than or equal

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_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 0 0 0 | Pg | Zn | 1 | Pd

Higher

CMPHI <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 0 1 0 0 size 0 | Zm 0 0 0 | Pg | Zn | 0 | Pd

Higher or same

CMPHS <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_GE;
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 size 0 | Zm 1 0 1 | Pg | Zn | 1 | Pd
Not equal

\texttt{CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>}

if \texttt{!HaveSVE() then UNDEFINED;}
integer \texttt{esize = 8 << Uint(size);} 
integer \texttt{g = Uint(Pg);} 
integer \texttt{n = Uint(Zn);} 
integer \texttt{m = Uint(Zm);} 
integer \texttt{d = Uint(Pd);} 
\texttt{SVECmp op = Cmp\_NE;} 
boolean \texttt{unsigned = FALSE;}

\textbf{Assembler Symbols}

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{size} & \texttt{T} \\
\hline
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\hline
\end{tabular}
\end{center}

\texttt{<Pg>} Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

\texttt{<Zn>} Is the name of the first source scalable vector register, encoded in the "Zn" field.

\texttt{<Zm>} Is the name of the second source scalable vector register, encoded in the "Zm" field.

\textbf{Operation}

\texttt{CheckSVEEnabled();}
integer \texttt{elements = VL DIV esize;}
bits(PL) \texttt{mask = P[g];}
bits(VL) \texttt{operand1 = Z[n];}
bits(VL) \texttt{operand2 = Z[m];}
bits(PL) \texttt{result;}
for \texttt{e = 0 to elements-1}
  integer \texttt{element1 = Int(Elem[operand1, e, esize], unsigned);}
  integer \texttt{element2 = Int(Elem[operand2, e, esize], unsigned);}
  if \texttt{ElemP[mask, e, esize] == '1' then}
    boolean \texttt{cond;}
    case \texttt{op of}
      when \texttt{Cmp\_EQ} \texttt{cond = element1 == element2;}
      when \texttt{Cmp\_NE} \texttt{cond = element1 != element2;}
      when \texttt{Cmp\_GE} \texttt{cond = element1 >= element2;}
      when \texttt{Cmp\_LT} \texttt{cond = element1 < element2;}
      when \texttt{Cmp\_GT} \texttt{cond = element1 > element2;}
      when \texttt{Cmp\_LE} \texttt{cond = element1 <= element2;}
        \texttt{ElemP[result, e, esize] = if cond then '1' else '0';}
      else
        \texttt{ElemP[result, e, esize] = '0';}
    \end{case}
  PSTATE.<N,Z,C,V> = \texttt{PredTest(mask, result, esize;}
P\texttt{d} = \texttt{result;}

\textbf{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.

\texttt{CMP\<cc> (vectors)}
The values of the NZCV flags.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  |

SVE

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.

<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] = 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 \texttt{MOVPRFX} instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is \texttt{UNPREDICTABLE}:

- 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.

The \texttt{MOVPRFX} instructions that can be used with this instruction are as follows:

- An unpredicated \texttt{MOVPRFX} instruction.
- A predicated \texttt{MOVPRFX} instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
<th>1</th>
<th>0</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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<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>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

CNT <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] = BitCount(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 0 1 0 0 | 0 1 0 | imm4 1 1 1 0 0 0 | pattern | Rd

Byte

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 0 1 0 0 | 1 1 1 0 | imm4 1 1 1 0 0 0 | pattern | Rd

Doubleword

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 0 1 0 0 | 1 1 0 | imm4 1 1 1 0 0 0 | pattern | Rd

Halfword

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;
Word

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>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10011</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>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

CheckSVEEnabled();
iinteger 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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
CNTP

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.

<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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

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;
bits(PL) mask = P[g];
bits(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  | 1  | 0  | 1  | 1  | sz | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | Pg | Zn | Zd |

SVE

COMPACT <Zd>, <T>, <Pg>, <Zn>, <T>

if !HaveSVE() then UNDEFINED;
integer esize = 32 << UInt(sz);
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 "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>

<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 ElemP[mask, e, esize] == '1' then
    bits(esize) element = Elem[operand1, e, esize];
    Elem[result, x, esize] = element;
    x = x + 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 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, 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 "#<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 alias **MOV (immediate, predicated, zeroing).**

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | Pg | 0  | 0  | sh | imm8 | Zd |

**SVE**

```
CPY <Zd>.<T>, <Pg>/Z, #<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

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 "#<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 aliases *FMOV (zero, predicated)*, and *MOV (immediate, predicated, merging)*.

```
31 30 29 28 27 26 25 24 23 22 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 sh | imm8 | Zd
```

**SVE**

```
CPY <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}
```

*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

```plaintext
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.

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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)**.

![Register Encoding](image)

*SVE*

**CPY <Zd>,<T>,<Pg>/M,<R><n|SP>**

if ![HaveSVE()](image) 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' then
        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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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
 0  0  0  0  0  1  0  1  size  1  0  0  0  0  1  0  0  Pg  Vn  Zd
```

**SVE**

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;
bite(P) mask = P[g];
bite(esize) operand1 = V[n];
bite(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | sz | 1 | Rm | 0 | 0 | 1 | 0 | 0 | 0 | Rn | 0 | 0 | 0 | 0 |
```

#### 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 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | sz | 1 | Rm | 0 | 0 | 1 | 0 | 0 | 0 | Rn | 1 | 0 | 0 | 0 |
```

#### 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

<\textcolor{red}{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

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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  |
```

**Byte**

```
DECB <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  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | 0  |
```

**Doubleword**

```
DECD <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  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  |
```

**Halfword**

```
DECH <Xdn>{, <pattern>{, MUL #<imm>}}
```

if !HaveSVE() then UNDEFINED;
integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;

DECB, DECD, DECH, DECW (scalar)
Word

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 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 | imm4 | 1 1 1 0 0 1 | pattern | Rdn |

Word

```
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;

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>&lt;pattern&gt;</th>
<th>00000</th>
<th>00001</th>
<th>00010</th>
<th>00011</th>
<th>00100</th>
<th>00101</th>
<th>00110</th>
<th>00111</th>
<th>01000</th>
<th>01001</th>
<th>01010</th>
<th>01011</th>
<th>01100</th>
<th>01101</th>
<th>0111x</th>
<th>10110</th>
<th>10111</th>
<th>1100x</th>
<th>11100</th>
<th>11110</th>
<th>11111</th>
</tr>
</thead>
<tbody>
<tr>
<td>POW2</td>
<td>VL1</td>
<td>VL2</td>
<td>VL3</td>
<td>VL4</td>
<td>VL5</td>
<td>VL6</td>
<td>VL7</td>
<td>VL8</td>
<td>VL16</td>
<td>VL32</td>
<td>VL64</td>
<td>VL128</td>
<td>VL256</td>
<td>#uimm5</td>
<td>#uimm5</td>
<td>#uimm5</td>
<td>#uimm5</td>
<td>#uimm5</td>
<td>#uimm5</td>
<td>#uimm5</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>
</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>
</tr>
</tbody>
</table>

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
icnt = DecodePredCount(pat, esize);
X[dn] = operand1 - (cnt * 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.
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**

```
0 0 0 0 0 1 0 0 1 1 1 0 0 0 1
   \__________________________
   \     \__________\           
   \     |imm4   |pattern|Zdn|
```

```
DECD <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**

```
0 0 0 0 0 1 0 0 1 1 1 0 0 0 0
   \__________________________
   \     \__________\           
   \     |imm4   |pattern|Zdn|
```

```
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**

```
0 0 0 0 0 1 0 0 1 0 1 1
   \__________________________
   \     \__________\           
   \     |imm4   |pattern|Zdn|
```

```
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.

<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

\[\text{CheckSVEEnabled}();\]
\[\text{integer elements} = \text{VL} \div \text{DIV esize};\]
\[\text{integer count} = \text{DecodePredCount}(\text{pat, esize});\]
\[\text{bits(VL) operand}1 = Z[\text{dn}];\]
\[\text{bits(VL) result};\]
\[\text{for e} = 0 \text{ to elements-1}\]
\[\quad \text{Elem}[\text{result, e, esize}] = \text{Elem}[\text{operand}1, e, \text{esize}] - (\text{count} \times \text{imm});\]
\[Z[\text{dn}] = \text{result};\]

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 registers.
  - The values of the NZCV flags.

- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  |

SVE

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(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;

Operational information

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to asynchronous exceptions does not vary based on:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
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 | size | 1 0 1 1 0 1 1 0 1 1 0 0 0 0 0 0 0 0 Pm | Zdn |

SVE

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.

<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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  |

SVE

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 0 1 0 1 | size 1 0 0 0 0 0 0 0 1 1 1 0 | Rn _______________ Zd
```

**SVE**

\[
\text{DUP} <Zd>.<T>, <R><n|SP>
\]

\[
\text{if } !\text{HaveSVE}() \text{ then UNDEFINED;}
\]

\[
\text{integer } \text{esize} = 8 \times \text{UInt}(\text{size});
\]

\[
\text{integer } n = \text{UInt}(\text{Rn});
\]

\[
\text{integer } d = \text{UInt}(\text{Zd});
\]

**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”:

  \[
  \begin{array}{c|c}
  \text{size} & <T> \\
  00 & B \\
  01 & H \\
  10 & S \\
  11 & D \\
  \end{array}
  \]

- **<R>** Is a width specifier, encoded in “size”:

  \[
  \begin{array}{c|c}
  \text{size} & <R> \\
  01 & W \\
  x0 & W \\
  11 & X \\
  \end{array}
  \]

- **<n|SP>** Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

**Operation**

\[
\text{CheckSVEEnabled}();
\]

\[
\text{integer } \text{elements} = \text{VL} \div \text{esize};
\]

\[
\text{bits(64) } \text{operand};
\]

\[
\text{if } n == 31 \text{ then}
\]

\[
\text{operand} = \text{SP}[];
\]

\[
\text{else}
\]

\[
\text{operand} = \text{X}[n];
\]

\[
\text{bits(\text{VL}) } \text{result};
\]

\[
\text{for } e = 0 \text{ to elements-1}
\]

\[
\text{Elem}[\text{result}, e, \text{esize}] = \text{operand}<\text{esize}-1:0>;
\]

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

**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.
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).

<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</td>
</tr>
</tbody>
</table>

SVE

DUP <Zd>.<T>, <Zn>.<T>[<imm>]

if !HaveSVE() then UNDEFINED;
bite(7) imm = imm2:tsz;
case tsz of
  when '00000' UNDEFINED;
  when '10000' esize = 128; index = UInt(imm<6:5>);
  when 'x1000' esize = 64;  index = UInt(imm<6:4>);
  when 'xx100' esize = 32;  index = UInt(imm<6:3>);
  when 'xxx10' esize = 16;  index = UInt(imm<6:2>);
  when 'xxxx1' 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.
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       Zd

SVE

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>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.

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.
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.

Assembly 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

\[
\text{CheckSVEEnabled();}
\]
\[
\text{bits}(\text{VL}) \text{ operand1} = \text{Z}\[\text{dn}\];
\]
\[
\text{bits}(\text{VL}) \text{ operand2} = \text{Z}\[\text{m}\];
\]
\[
\text{bits}(\text{VL}) \text{ operand3} = \text{Z}\[\text{k}\];
\]
\[
\text{Z}\[\text{dn}\] = \text{operand1} \text{ 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

| 31 30 29 28 27 26 25 24 23 22 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 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Pm | Pg | Pn | Pd |

Not setting the condition flags


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 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Pm | Pg | Pn | Pd |

Setting the condition flags


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

```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 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  |

**SVE**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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**.

```
31 30 29 28 27 26 25 24 23 22 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 1 0 0 0 0 imm13 Zdn
```

**SVE**

**EOR <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>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxxx</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>xxxxxxxx</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**

```plaintext
CheckSVEEnabled();
iinteger 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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 0 0 0 1 0 0 1 0 1 | Zm | 0 0 1 1 0 0 | Zn | Zd |

SVE

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.

SVE2

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.
<T> Is the 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE.
The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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.

<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>

SVE2

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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 1 0 0 | size | 0 1 1 0 0 1 0 1 | Pg | Zn | Vd

SVE

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**

| 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 | imm8h | 0 0 0 | imm8l | Zn | Zd |

**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 | imm8h | 0 0 0 | imm8l | Zm | Zdn |

**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

\[
\text{CheckSVEEnabled}(); \\
\text{integer elements} = \frac{\text{VL}}{\text{DIV esize}}; \\
\text{bits(}\text{VL}) \text{ operand1} = \text{Z}[s1]; \\
\text{bits(}\text{VL}) \text{ operand2} = \text{Z}[s2]; \\
\text{bits(}\text{VL}) \text{ result}; \\
\text{if position} \geq \text{elements then} \\
\quad \text{position} = 0; \\
\text{position} = \text{position} \ll 3; \\
\text{bits(}\text{VL}\times2) \text{ concat} = \text{operand2} : \text{operand1}; \\
\text{result} = \text{concat}\langle\text{position+}\frac{\text{VL}}{2}\text{-1:position}\rangle; \\
\text{Z}[dst] = \text{result};
\]

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.
FABD

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  |

SVE

FABD <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 Elem[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 | size | 0 1 1 1 0 0 1 0 1 | Pg | Zn | Zd |

SVE

FABS <Zd>..<T>, <Pg>/M, <Zn>..<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
ing = UInt(Pg);
n = UInt(Zn);
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] = FPAbs(element);

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | 1  | 0  | 1  | Zm | 1  | 1  | 1  | Pg | Zn | 1  | Ph |

Greater than

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);
ineger n = UInt(Zn);
inieger m = UInt(Zm);
inieger 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  | 1  | 1  | 0  | Zm | 1  | 1  | 0  | Pg | Zn | 1  | Ph |

Greater than or equal

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
inieger esize = 8 << UInt(size);
inieger g = UInt(Pg);
inieger n = UInt(Zn);
inieger m = UInt(Zm);
inieger 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.

FAC<cc>
<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(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);
        ElemP[result, e, esize] = if res then '1' else '0';
        else
            ElemP[result, e, esize] = '0';
    P[d] = result;
```
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.

| 31 30 29 28 27 26 25 24 23 22 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 | size | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Pg | 0 | 0 | 0 | 0 | i1 | Zdn |

SVE

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] = FPAdd(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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>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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

FADD <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 Elem[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE

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;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

<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>

SVE

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

<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>

<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>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;
FADDP

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  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |

SVE2

FADDP \(<Zdn>\.\<T>\), \<Pg>/M, \<Zdn>\.\<T>\), \<Zm>\.\<T>\)

if !\(\text{HaveSVE2}\)() then UNDEFINED;
if size == '00' 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>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

\(\text{CheckSVEEnabled}\)();
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 = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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>size</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

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);
**FCADD**

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.

```
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | size: | 0 | 0 | 0 | 0 | 0 | 0 | rot: 1 | 0 | 0 | Pg | Zm | Zdn |
```

**SVE**

```
FCADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>, <const>
```

```c
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);
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>0</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

\textbf{CheckSVEEnabled}();
integer pairs = \texttt{VL} \texttt{DIV} (2 * \texttt{esize});
bits(PL) mask = \texttt{P}[g];
bits(\texttt{VL}) operand1 = \texttt{Z}[dn];
bits(\texttt{VL}) operand2 = \texttt{Z}[m];
bits(\texttt{VL}) result;

for p = 0 to pairs-1
\begin{align*}
\text{acc}_r &= \texttt{Elem}[\text{operand1}, 2 \times p + 0, \text{esize}]; \\
\text{acc}_i &= \texttt{Elem}[\text{operand1}, 2 \times p + 1, \text{esize}]; \\
\text{elt2}_r &= \texttt{Elem}[\text{operand2}, 2 \times p + 0, \text{esize}]; \\
\text{elt2}_i &= \texttt{Elem}[\text{operand2}, 2 \times p + 1, \text{esize}]; \\
\text{if} \ E\text{lem}[\text{mask}, 2 \times p + 0, \text{esize}] &= '1' \text{ then} \\
&\quad \text{if sub}_i \text{ then} \text{elt2}_i = \texttt{FPNeg}(\text{elt2}_i); \\
&\quad \text{acc}_r = \texttt{FPAdd}(\text{acc}_r, \text{elt2}_i, \text{FPCR}); \\
\text{if} \ E\text{lem}[\text{mask}, 2 \times p + 1, \text{esize}] &= '1' \text{ then} \\
&\quad \text{if sub}_r \text{ then} \text{elt2}_r = \texttt{FPNeg}(\text{elt2}_r); \\
&\quad \text{acc}_i = \texttt{FPAdd}(\text{acc}_i, \text{elt2}_r, \text{FPCR}); \\
&\texttt{Elem}[\text{result}, 2 \times p + 0, \text{esize}] = \text{acc}_r; \\
&\texttt{Elem}[\text{result}, 2 \times p + 1, \text{esize}] = \text{acc}_i;
\end{align*}

\texttt{Z}[dn] = \text{result};

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is \textbf{UNPREDICTABLE}:

\begin{itemize}
  \item The 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}

The MOVPRFX instructions that can be used with this instruction are as follows:

\begin{itemize}
  \item An unpredicated MOVPRFX instruction.
  \item A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
\end{itemize}
**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

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

#### Equal

**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

#### Greater than

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

**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

#### Greater than or equal

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

**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

```
FCMLT <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_LT;

Less than or equal

```
FCMLE <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_LE;

Not equal

```
FCMNE <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_NE;

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 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(PL) result;
for e = 0 to elements-1
   bits(esize) element = Elem[operand, e, esize];
   if ElemP[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);
         ElemP[result, e, esize] = if res then '1' else '0';
      else
         ElemP[result, e, esize] = '0';
   end case;
end for;
P[d] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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

31 30 29 28 27 26 25 24 23 22 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 0 Pd

Equal

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);
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 Zm 0 1 0 Pg Zn 1 Pd

Greater than

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);
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 Zm 0 1 0 Pg Zn 0 Pd
Greater than or equal

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

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

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

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_NE res = FPCompareNE(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';
  end;

P[d] = result;
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Zm | 0  | rot | Pg | Zn | Zda |

SVE

FCMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>, <const>

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);
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.

<T> Is the 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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

```plaintext

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

```plaintext
FCMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <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

CheckSVEEnabled();
integer pairs = VL DIV (2 * esize);
integer pairspersegment = 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 pairspersegment;
    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 * p + 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
FCPY

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 register remain unmodified.

This instruction is used by the alias FMOV (immediate, predicated).

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the
following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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**

\[
\begin{array}{cccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & P_g & Z_n & Z_d
\end{array}
\]

**Half-precision to single-precision**

FCVT \(<Z_d>.S, <P_g>/M, <Z_n>.H\)

if !HaveSVE() then UNDEFINED;
integer esize = 32;
integer g = UInt(P_g);
integer n = UInt(Z_n);
integer d = UInt(Z_d);
integer s_esize = 16;
integer d_esize = 32;

**Half-precision to double-precision**

\[
\begin{array}{cccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & P_g & Z_n & Z_d
\end{array}
\]

**Half-precision to double-precision**

FCVT \(<Z_d>.D, <P_g>/M, <Z_n>.H\)

if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer g = UInt(P_g);
integer n = UInt(Z_n);
integer d = UInt(Z_d);
integer s_esize = 16;
integer d_esize = 64;

**Single-precision to half-precision**

\[
\begin{array}{cccccccccccccccccc}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & P_g & Z_n & Z_d
\end{array}
\]

**Single-precision to half-precision**

FCVT

Page 1558
Single-precision to half-precision

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

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

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

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 Elem[mask, e, esize] == '1' then
        bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR);
        Elem[result, e, esize] = ZeroExtend(res);

Z[d] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | Pg | Zn | Zd |

Half-precision to single-precision

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 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | Pg | Zn | Zd |

Single-precision to double-precision

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;
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 | Pg              | Zn              |
| 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 | Pg              | Zn              |
| 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);
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.

Double-precision to single-precision

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;

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) 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
**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 1 0 1 | Pg | Zn | Zd

**Double-precision to single-precision**

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

```
0 1 1 0 0 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1 0 1
Pg  Zn  Zd
```

### Half-precision to 16-bit

```
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

```
0 1 1 0 0 1 0 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 1
Pg  Zn  Zd
```

### Half-precision to 32-bit

```
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

```
0 1 1 0 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1
Pg  Zn  Zd
```
### Half-precision to 64-bit

FCVTZS <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 = 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   1   1   1   1   1   1   0   1   1   1   0   1   1   1   1   1   1   1   0   1   1   0   1   0   1   0   1 |

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 = 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   1   0   1   1   1   1   1   0   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   0   1   1   0   1   0   1 |

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 = 32;
integer d_esize = 64;
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   1   0   1   1   1   0   0   1   0   1   1   0   1   1   0   0   1   0   1   0   1 |

FCVTZS

Page 1568
Double-precision to 32-bit

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

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

<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) res = FPToFixed(element<s_esize-1:0>, 0, unsigned, FPCR, rounding);
      Elem[result, e, esize] = Extend(res, unsigned);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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

```
<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>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>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>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Half-precision to 16-bit

```
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

```
<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>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>1</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Half-precision to 32-bit

```
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

```
<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>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>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>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Half-precision to 64-bit
Half-precision to 64-bit

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 |
|---------------------------------|----------------|----------------|
|                                | Pg             | Zn             | Zd             |

Single-precision to 32-bit

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 |
|--------------------------------|----------------|----------------|
|                                      | Pg             | Zn             | Zd             |

Single-precision to 64-bit

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 |
|--------------------------------|----------------|----------------|
|                                      | Pg             | Zn             | Zd             |
Double-precision to 32-bit

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

FCVTZU <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 = FPRounding_ZERO;

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) res = FPToFixed(element<s.esize-1:0>, 0, unsigned, FPCR, rounding);
        Elem[result, e, esize] = Extend(res, unsigned);

Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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>

SVE

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);

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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 | Pg              | Zm              | Zdn              |

SVE

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.

<T> Is the 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 Elem[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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).

| 31 30 29 28 27 26 25 24 23 22 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 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | imm8 | | Zd |

SVE

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  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  |

**SVE**

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] = FPExA(element);
Z[d] = result;

---

Copyright © 2010-2019 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.

```
31 30 29 28 27 26 25 24 23 22 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 0 1 1  size 0 1 0 1 Pg Zn Zd
```

SVE2

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 Elem[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 MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
FMAD

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | Pg  | Zm  | Zdn |

SVE

FMAD <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 = FALSE;
boolean op3_neg = 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>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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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 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  | 0  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| size | 0  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | Pg  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE

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);
bites(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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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>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>1</td>
<td>0</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>
</tr>
</tbody>
</table>

SVE

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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

```
| 31 30 29 28 27 26 25 24 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     |
| size           | Pg             | i1             | Zdn             |
```

SVE

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(P[<T>]) 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 size 0 0 0 1 0 0 1 0 0 Pg Zm Zdn

SVE

FMAXNM <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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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            | Pg              |
| 0 1 0 1 0 0     | Zm              | Zdn             |

**SVE2**

FMAXNMP `<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>`

```plaintext
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**

```plaintext
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[operand2, e + 1, esize];
    else
      element1 = Elem[operand2, e - 1, esize];
      element2 = Elem[operand1, e + 0, esize];
    end
  Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR);
Z[dn] = result;
```

FMAXNMP
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 0  | 0  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |

SVE

FMAXNMV <V><d>, <Pg>, <Zn>.<T>

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
to integer esize = 8 << Uint(size);
to integer g = Uint(Pg);
to integer n = Uint(Zn);
to 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);

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

<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>1</td>
<td>0</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></td>
<td></td>
</tr>
</tbody>
</table>

**SVE2**

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**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | Pg | Zn | Vd |

SVE

FMAXV <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('1');

V[d] = ReducePredicated(ReduceOp_FMAX, operand, mask, identity);

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 1 0 0 Pg | 0 0 0 0 | i1 | Zdn

**SVE**

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);
bites(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;
bites(Pg) mask = P[g];
bites(VL) operand1 = Z[dn];
bites(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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
```

SVE

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(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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

```
0 1 1 0 0 1 0 1 1 0 1 1 0 0 1 0 0 0 0 |1| 0 1 1 0 0 1 0 0 0 1 1 0 1 0 0 1 1 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
```

**SVE**

```
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#0.0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>#1.0</td>
<td></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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | Pg | Zm | Zdn|
```

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
FMINNMP

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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |

SVE2

FMINNMP <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] = FPMinNum(element1, element2, FPCR);
Z[dn] = result;
Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 |

SVE

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>&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>

<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_FMINNUM, operand, mask, identity);
**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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 0  | Pg | Zm | Zdn |

**SVE2**

FMINP `<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] = FPMin(element1, element2, FPCR);

Z[dn] = result;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

### SVE

FMINV <V>, <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 \([Zda = Zda + Zn \ast Zm]\).

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 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 size 1 Zm 0 0 0 Pg Zn Zda

SVE

FMLA <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 = 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
FMLA (indexed)

Floating-point fused multiply-add by indexed elements (Zda = Zda + Zn * Zm[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

```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  1  0  0  1  0  0  0  |i3h| 1  |i3l| Zm  0  0  0  0  0  0  Zn  Zda
```

### Single-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  1  0  0  1  0  0  1  | 0  1| i2 | Zm  0  0  0  0  0  0  Zn  Zda
```

### 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  1  0  0  1  0  0  1  | 1  1| 1  | Zm  0  0  0  0  0  0  Zn  Zda
```

if !HaveSVE() then UNDEFINED;  
integer esize = 64;  
integer index = UInt(il);  
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:  
  • The MOVPRFX instruction must specify the same destination register as this instruction.  
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.  
The MOVPRFX instructions that can be used with this instruction are as follows:  
  • An unpredicated MOVPRFX 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.

Internals

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE2


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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 0 1 0 0 1 0 1 0 3h  Zm 0 1 0 0 i3l 0  Zn  Zda
```

Single-precision


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 + 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX 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 |

SVE2


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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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.

<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>

Single-precision


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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

```
0 1 1 0 0 1 0 1  size 1 | Zm 0 0 1  | Pg | Zn | Zda
```

SVE

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

```
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09 rc2_1, sve v2019-09 rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
FMLS (indexed)

Floating-point fused multiply-subtract by indexed elements ($Zda = Zda + -Zn * Zm[indexed]$).

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 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

| 31 30 29 28 27 26 25 24 23 22 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 | i3h | i3l | Zm 0 0 0 0 0 1 | Zn | Zda |

Half-precision

```c
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

| 31 30 29 28 27 26 25 24 23 22 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 | i2  | Zm 0 0 0 0 0 1 | Zn | Zda |

Single-precision

```c
FMLS <Zda>.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 da = UInt(Zda);
boolean op1_neg = TRUE;
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 | i1  | Zm 0 0 0 0 0 1 | Zn | Zda |
Double-precision


if !HaveSVE() then UNDEFINED;
integer esize = 64;
integer index = UInt(il);
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
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.

<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 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

SVE2


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();
intrange 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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.

### Single-precision

\[
\text{FMLSLB } <\text{Zda}>.S, <\text{Zn}>.H, <\text{Zm}>.H[^{\text{imm}}]
\]

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX 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.

```
0 1 1 0 0 1 0 0 1 | Zm
1 0 1 0 0 1 | Zn
-----|-----
0 1 1 0 0 1 0 0 1 | Zda
```

SVE2


if !HaveSVE2() then UNDEFINED;

integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;

Assemble 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) operands1 = Z[n];
bits(VL) operands2 = Z[m];
bits(VL) operands3 = Z[da];
bits(VL) result;
```

```
for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operands1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operands2, 2 * e + 1, esize DIV 2];
    bits(esize) element3 = Elem[operands3, 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

### Single-precision

**FMLSLT** <Zda>.S, <Zn>.H, <Zm>.H[

```plaintext
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**

```plaintext
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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 30 29 28 27 26 25 24 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</td>
</tr>
<tr>
<td>1 1 1 0 0 1</td>
</tr>
<tr>
<td>Zda</td>
</tr>
</tbody>
</table>

32-bit element

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 30 29 28 27 26 25 24 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 1 1</td>
</tr>
<tr>
<td>1 1 1 0 0 1</td>
</tr>
<tr>
<td>Zda</td>
</tr>
</tbody>
</table>

64-bit element

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 < esize * 4 then UNDEFINED;
integer segments = VL DIV (4 * esize);
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) operand3 = Z[da];
bits(VL) result = Zeros();
bits(4*esize) op1, op2;
bits(4*esize) res, addend;

for s = 0 to segments-1
    op1 = Elem[operand1, s, 4*esize];
    op2 = Elem[operand2, s, 4*esize];
    addend = Elem[operand3, s, 4*esize];
    res = FPMatMulAdd(addend, op1, op2, esize, FPCR);
    Elem[result, s, 4*esize] = res;

Z[da] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
FMSB

Floating-point fused multiply-subtract vectors (predicated), writing multiplicand \(Z_{dn} = Za + (-Z_{dn} \times Zm)\).

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  | size| 1  | 0  | 1  | 0  | 1  | Pg | Zm | Zdn |

SVE

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.

<T> Is the 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 30 29 28 27 26 25 24 23 22 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 0 0 1 0 0 1 0 0 | Pg 0 0 0 0 | i1 | Zdn |

**SVE**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 1 0 1 size | 0 0 0 1 0 1 0 0 | Pg | Zm | Zdn

SVE

FMUL <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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | size | Zm | 0  | 0  | 0  | 1  | 0  | Zn | Zd |

SVE

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

```
0 1 1 0 0 1 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

```
0 1 1 0 0 1 0 0 |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

```
0 1 1 0 0 1 0 0 |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.

<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;
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;
**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 ∞×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.

```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  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |
| size | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | Pg | Zm | Zdn |
```

**SVE**

```
FMULX <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] = FPMulX(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
    register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
Floating-point negated 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 negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.

### SVE

SVE

\[
\text{FNMAD } <Z_{dn}>, <T>, <Pg>/M, <Z_m>., <T>, <Z_a>.
\]

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

- **<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.
- **<Z_m>** 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

\[
\text{CheckSVEEnabled}();
\]

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

\[
\begin{align*}
\text{bits}(esize) \text{ element1} &= \text{Elem}[\text{operand1}, e, esize]; \\
\text{bits}(esize) \text{ element2} &= \text{Elem}[\text{operand2}, e, esize]; \\
\text{bits}(esize) \text{ element3} &= \text{Elem}[\text{operand3}, e, esize]; \\
\end{align*}
\]

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
Floating-point negated fused multiply-add vectors (predicated), writing addend \(Zda = -Zda + -Zn \times Zm\).

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

\(<Zda>\) 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
Floating-point negated 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 negated 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   | Zm  | 0   | 1   | 1   | Pg  | Zn  | Zda |

SVE

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

```c
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;
  end if

Z[da] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand $[Zdn = -Za + Zdn \times Zm]$.

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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | | Za | 1 | 1 | 1 | Pg | Zm | Zdn |

SVE

FNMSB $<Zdn>.<T>$, $<Pg>/M$, $<Zm>.<T>$, $<Za>.<T>$

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 = FALSE;
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

* The MOVPRFX instruction must specify the same destination register as this instruction.
* The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

* An unpredicated MOVPRFX instruction.
* A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

```
| 31 30 29 28 27 26 25 24 23 22 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 1 0 0 1 1 0 0 | Zn | Zd |
```

**SVE**

FRECPE <Zd>,<T>, <Zn>,<T>

```c
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”:
  - size <T>
  - 00 | RESERVED
  - 01 | H
  - 10 | S
  - 11 | D
- `<Zn>` Is the name of the source scalable vector register, encoded in the “Zn” field.

**Operation**

```c
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 © 2010-2019 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 1  | 0  | 1  | 0  | 2  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**SVE**

FRECPS `<Zd>..<T>`, `<Zn>..<T>`, `<Zm>..<T>`

if `!HaveSVE()` 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><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.

**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] = FPRecipStepFused(element1, element2);
- Z[d] = result;
Floating-point reciprocal exponent (predicated).

Invert the exponent leaving the fractional part unchanged 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 FMULX to convert arbitrary elements in mathematical vector space to "unit vectors" or "direction vectors" of length 1.

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

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) 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
**FRINT<r>**

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 | 1 0 1 | Pg | Zn | Zd |

**Current mode signalling inexact**

\[ \text{FRINTI } \langle Zd \rangle, \langle T \rangle, \langle Pg / M \rangle, \langle Zn \rangle, \langle T \rangle \]

\[
\text{if } \text{!HaveSVE}() \text{ then UNDEFINED;}
\text{if size == '00' then UNDEFINED;}
\text{integer esize = 8 << UInt(size);} \\
\text{integer g = UInt(Pg);} \\
\text{integer n = UInt(Zn);} \\
\text{integer d = UInt(Zd);} \\
\text{boolean exact = FALSE;}
\text{FPRounding rounding = FPRoundingMode(FPCR);} \\
\]

**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 | 0 0 | 1 0 | 1 0 | Pg | Zn | Zd |
Nearest with ties to away

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 |
|------------------------|--------|--------|--------|
| size                  | Pg     | Zn     | Zd     |
| 0 1 1 0 0 1 0 1 0 1    | 0 0 0 0 0 0 1 0 1                  |

Nearest with ties to even

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 |
|------------------------|--------|--------|--------|
| size                  | Pg     | Zn     | Zd     |
| 0 1 1 0 0 1 0 1 0 1    | 0 0 0 0 1 1 0 1                  |

Toward zero

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 |
|------------------------|--------|--------|--------|
| size                  | Pg     | Zn     | Zd     |
| 0 1 1 0 0 1 0 1 0 1    | 0 0 0 1 0 1 0 1                  |
Toward minus infinity

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 0 1 1 0 1 |
| Pg                             | Zn  | Zd    |

Toward plus infinity

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(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] = FPRoundInt(element, FPCR, rounding, exact);
Z[d] = result;

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE.
The MOVPRFX instruction must specify the same destination register as this instruction. The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 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| 1| 1| 0| 0| 1| 1| 0| 0| 0| 0| 1| 0| 0| 0| 0| 1| 1| 0| 0| Zn| Zd|

SVE

**FRSQRTE** <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": |
| size | <T> |
| 00   | RESERVED |
| 01   | H        |
| 10   | S        |
| 11   | D        |

| <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] = FPRSqrtEstimate(element, FPCR);
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 |

SVE

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;
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] = 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 0 1 1 0 0 1 1 0 0 0 0 1 0 1 0 0 1 0 0 1
```

Sve

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

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 Elem[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
FSQRT

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 |

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | Zdn |

SVE

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 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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 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   | 0   | 0   | 0   | 1   | 1   | 0   | 0   | Pg  | Zm  | Zdn |

**SVE**

FSUB <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(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

```
0 1 1 0 0 1 0 1 0 0
Zm 0 0 0 0 1 0
Zn 0 0
Zd
```

SVE

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;
**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.

<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>

**SVE**

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>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(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 30 29 28 27 26 25 24 23 22 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 0 | Pg | Zm | Zdn |

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 \(-\pi/4 < x \leq \pi/4\).

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | Zm | Zdn|

**SVE**

**FTMAD** \(<Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<imm>\)

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer imm = UInt(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 “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 unsigned 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 \((2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4\).

SVE

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.

<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: \((2q-1) \times \pi/4 < x \leq (2q+1) \times \pi/4\).

| 31 30 29 28 27 26 25 24 23 22 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            | Zm               | 1 0 1 1 0 0      | Zn               | Zd               |

SVE

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;

Copyright © 2010-2019 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 0 1 0 1 | size | 1 | Zm | 1 1 0 | Pg | Zn | Zd |

SVE2

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;

Operational information

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.
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 1 0 1 1 |          size   | 1              | Zm             | 1 0 1 0 0 0    | Zn             | Zd             |

SVE2

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;

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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 0 1 0 0 | 0 1 1 | imm4 | 1 1 1 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 0 1 0 0 | 1 1 1 | imm4 | 1 1 1 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 0 1 0 0 | 1 1 1 | imm4 | 1 1 1 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>}}

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.
<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>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10011</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11100</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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
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 0 1 0 0 | 1 | 1 | 1 | imm4 | 1 | 1 | 0 | 0 | 0 | 0 | pattern | Zdn

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

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 | 1 | imm4 | 1 | 1 | 0 | 0 | 0 | 0 | pattern | Zdn

INCD <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;

INCH <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;

INCW <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>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
   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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |

SVE

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;

Operational information

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

<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>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>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>

SVE

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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SVE

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.

SVE

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 0 1 0 0 | size | 1 | imm5 | 0 1 0 0 0 1 | Rn | Zd |

SVE

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>10</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);
bits(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.

<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>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>0</td>
<td>0</td>
<td>1</td>
<td>0</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></td>
<td></td>
</tr>
</tbody>
</table>

SVE

INDEX <Zd>,<T>, <R><n>, <R><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
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>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.

Operation

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;

Operational information

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to 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 |
|-----------------|-----------------|-----------------|
| size            | 1               | Rm              |
| Zdn             | 1               |
```

**SVE**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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.

31 30 29 28 27 26 25 24 23 22 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 1 0 0 0 1 1 1 0 | Vm | Zdn

**SVE**

INSR <Zdn>,<T>, <V><m>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Vm);

**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>

<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>

<m> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vm" field.

**Operation**

CheckSVEEnabled();
bits(VL) dest = Z[dn];
bits(esize) src = V[m];
Z[dn] = dest<VL-esize-1:0> : src;

**Operational information**

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

```

SVE

LASTA <R><d>, <Pg>, <Zn>.

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

<table>
<thead>
<tr>
<th>&lt;R&gt;</th>
<th>Is a width specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</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. |

<table>
<thead>
<tr>
<th>&lt;T&gt;</th>
<th>Is the size specifier, encoded in “size”:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</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>

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 (scalar)
Operational information

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 © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  |

SVE

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”:

<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];

Operational information

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.
**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.

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

**SVE**

LASTB <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 = TRUE;

**Assemble Symbols**

<\textit{R}> Is a width specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\textit{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>

<\textit{d}> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<\textit{Pg}> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<\textit{Zn}> Is the name of the source scalable vector register, encoded in the "Zn" field.

<\textit{T}> Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>&lt;\textit{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;
Operational information

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.
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.

SVE

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];

Operational information

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.
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 read 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 0 1 0 0 0 1             | imm5  | 1 1 0 |
|                                  |       | Pg    | Zn    |
|                                  |       |       | Zt    |

32-bit element

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    |

64-bit element

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 \text{ DIV} esize;
bids(VL) base = Z[n];
bids(64) addr;
bids(PL) mask = P[g];
bids(VL) result;
bids(msize) data;
constant integer mbytes = msize \text{ DIV} 8;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1' then
    \text{addr} = \text{ZeroExtend}(\text{Elem}[\text{base, e, esize}], 64) + \text{offset} * \text{mbytes};
    \text{data} = \text{Mem}[	ext{addr, mbytes, AccType NORMAL}];
    \text{Elem}\,[\text{result, e, esize}] = \text{Extend}(\text{data, esize, unsigned});
  else
    \text{Elem}\,[\text{result, e, esize}] = \text{Zeros}();
\text{Z}[t] = \text{result};
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 read 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 1 0 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

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

<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 1 0 0 0 1 0 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

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

<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 1 0 0 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>
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

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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 read 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>, <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

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

LD1B (scalar plus scalar)
32-bit element

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 | Rm | 0 1 0 | Pg | Rn | Zt |
|--------------------------------------------------------------------------------------|
| 1 0 1 0 0 1 0 0 1 0 1 1 |                                                                                     |

64-bit element

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 HaveMTEEExt() then SetNotTagCheckedInstruction(FALSE);

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

for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    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();
    offset = offset + 1;

Z[t] = result;
LD1B (scalar plus vector)

Gather load unsign bytes to vector (vector index).

Gather load of unsign 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 read 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

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

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

LD1B (scalar plus vector)
LD1B { <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 = 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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
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 read 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   1   0   1   1   0   1   imm5  1   1   0   Pg   Zn   Zt</td>
</tr>
</tbody>
</table>

SVE

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);

Assember 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 HaveMTEEExt() then SetNotTagCheckedInstruction(FALSE);
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 read 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  | 1  | 1  | 1  | 0  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

SVE

LD1D { <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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
**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 read 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  | 1  | 0  | 1  | 1  | 1  | 1  | Rm |   0 | 1  | 0  |   Pg | Rn |   Zt |   |

SVE

```
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**

```
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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
```
LD1D (scalar plus scalar)
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 read 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

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

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 = 0;

64-bit scaled offset

LD1D (scalar plus vector)
64-bit scaled offset

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

| 31 30 29 28 27 26 25 24 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 0 | Zm | 1 1 0 | Pg | Rn | Zt |

64-bit unscaled offset

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 "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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
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 read 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</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>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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32-bit element</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDH { &lt;Zt&gt;.S }, &lt;Pg&gt;/Z, [&lt;Zn&gt;.S{, #&lt;imm&gt;}]</td>
</tr>
<tr>
<td>if !HaveSVE() then UNDEFINED;</td>
</tr>
<tr>
<td>integer t = UInt(Zt);</td>
</tr>
<tr>
<td>integer n = UInt(Zn);</td>
</tr>
<tr>
<td>integer g = UInt(Pg);</td>
</tr>
<tr>
<td>integer esize = 32;</td>
</tr>
<tr>
<td>integer msize = 16;</td>
</tr>
<tr>
<td>boolean unsigned = TRUE;</td>
</tr>
<tr>
<td>integer offset = UInt(imm5);</td>
</tr>
</tbody>
</table>

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>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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>64-bit element</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDH { &lt;Zt&gt;.D }, &lt;Pg&gt;/Z, [&lt;Zn&gt;.D{, #&lt;imm&gt;}]</td>
</tr>
<tr>
<td>if !HaveSVE() then UNDEFINED;</td>
</tr>
<tr>
<td>integer t = UInt(Zt);</td>
</tr>
<tr>
<td>integer n = UInt(Zn);</td>
</tr>
<tr>
<td>integer g = UInt(Pg);</td>
</tr>
<tr>
<td>integer esize = 64;</td>
</tr>
<tr>
<td>integer msize = 16;</td>
</tr>
<tr>
<td>boolean unsigned = TRUE;</td>
</tr>
<tr>
<td>integer offset = UInt(imm5);</td>
</tr>
</tbody>
</table>

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(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    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;
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 read 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

```
| 31 30 29 28 27 26 25 24 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 1 0 1 0 imm4 1 0 1 Pg Rn Zt |
```

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

```
| 31 30 29 28 27 26 25 24 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 0 1 Pg Rn Zt |
```

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

```
| 31 30 29 28 27 26 25 24 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 1 0 imm4 1 0 1 Pg Rn Zt |
```

64-bit element

```
LD1H (scalar plus immediate)
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
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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;
**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 read 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</td>
</tr>
</tbody>
</table>

**16-bit element**

LD1H \{ \langle Zt \rangle.H \}, \langle Pg \rangle/Z, [\langle Xn|SP \rangle, \langle Xm \rangle, 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 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>

**32-bit element**

LD1H \{ \langle Zt \rangle.S \}, \langle Pg \rangle/Z, [\langle Xn|SP \rangle, \langle Xm \rangle, 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 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>

**64-bit element**

LD1H (scalar plus scalar)
64-bit element

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 read 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

```
1 0 0 0 0 1 0 1 0 0 1 xs 1 Zm 0 1 0 Pg Rn Zt
```

### 32-bit scaled offset

```java
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

```
1 1 0 0 0 1 0 1 0 0 1 xs 1 Zm 0 1 0 Pg Rn Zt
```

### 32-bit unpacked scaled offset

```java
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

```
1 1 0 0 1 0 0 1 xs 0 Zm 0 1 0 Pg Rn Zt
```

### 32-bit unpacked unscaled offset

```java
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 unscaled offset

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

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

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

LD1H (scalar plus vector)
64-bit unscaled offset

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 then
    CheckSPAlignment();
    base = SP[];
else
    base = X[n];
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1'
        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;
LD1RB

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | imm6 | 1 | 0 | 0 | Pg | Rn | Zt |

#### 8-bit element

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;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

### 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  | 0  | 0  | 1  | 0  | 0  | 1  | imm6 | 1 | 0 | 1 | Pg | Rn | Zt |

#### 16-bit element

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;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);

### 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  | imm6 | 1 | 1 | 0 | Pg | Rn | Zt |
32-bit element

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;
integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm);

64-bit element

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

```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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
```
**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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | imm6| 1  | 1  | 1  | Pg | Rn | Zt |

**SVE**

LD1RD \(\{<Zt>.D\},<Pg>/Z,\{<Xn|SP>{, #<imm>}\})

if \(!\text{HaveSVE}()\) then UNDEFINED;

integer \(t = \text{UInt}(Zt);\)

integer \(n = \text{UInt}(Rn);\)

integer \(g = \text{UInt}(Pg);\)

integer esize = 64;

integer msize = 64;

boolean unsigned = TRUE;

integer offset = \(\text{UInt}(\text{imm6});\)

**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.
- \(<\text{imm}>\) 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.
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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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

```
<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 1 1 imm6 1 0 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>
```

### 16-bit element

```
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

```
<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 1 1 imm6 1 1 0 Pg Rn Zt</td>
</tr>
</tbody>
</table>
```

### 32-bit element

```
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

```
<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 1 1 imm6 1 1 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>
```
64-bit element

LD1RH \{ <Zt>.D \}, \langle Pg\rangle/Z, [<Xn\langle SP\rangle\{} , #<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\langle SP\rangle> 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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 read 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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | imm4 | 0  | 0  | 1  | Pg | Rn  | Zt  |
```

**SVE**

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;

\( Z[t] = \text{ZeroExtend(Replicate(result, VL, DIV 256), VL);} \)
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 read 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.

ID_AA64ZFR0_EL1.F64MM

SVE

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

```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(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 read 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.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 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   | 0   | imm4| 0   | 0   | 1   | Pg  | Rn  | Zt  |

**SVE**

```plaintext
LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}
```

if !HaveSVEFP64MatMulExt() 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 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.
Operation

CheckSVEEnabled();
if \text{VL} < 256 then UNDEFINED;
integer elements = 256 \text{DIV} esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g]; // low bits only
bits(256) result;
constant integer mbytes = esize \text{DIV} 8;

if n == 31 then
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  base = X[n];

addr = base + offset * 32;
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;
\text{Z[t]} = ZeroExtend(Replicate(result, VL \text{ 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 read 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.

**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 SetNotTagCheckedInstruction(FALSE);

if $n == 31$ 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;

$Z[t] = ZeroExtend(Replicate(result, VL DIV 256), VL)$;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 read 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.

```
31 30 29 28 27 26 25 24 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 | imm4 | 0 0 1   | Pg | Rn | Zt
```

SVE

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 \( \text{esize} \);
bits(64) base;
bits(64) addr;
bits(PL) mask = \( P[g] \); // low bits only
bits(256) result;
constant integer mbytes = \( \text{esize} \) DIV 8;
if \( n == 31 \) then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = \( SP[] \);
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    base = \( X[n] \);
addr = base + offset * 32;
for \( e = 0 \) to elements-1
    if \( \text{ElemP}[\text{mask}, e, \text{esize}] == '1' \) then
        \( \text{Elem}[	ext{result}, e, \text{esize}] = \text{Mem}[\text{addr}, \text{mbytes}, \text{AccType_NORMAL}] \);
    else
        \( \text{Elem}[	ext{result}, e, \text{esize}] = \text{Zeros}() \);
    addr = addr + mbytes;
\( Z[t] = \text{ZeroExtend}(	ext{Replicate}(	ext{result, VL DIV 256}, VL)) \);
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 read 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.

```
<table>
<thead>
<tr>
<th>Rm</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

SVE

```
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 SetNotTagCheckedInstruction(FALSE);

if \( n == 31 \) 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), \( VL \));
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 read 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  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | Pg | Rn | Zt |

SVE

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);

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;

\( Z[t] = ZeroExtend(Replicate(result, VL \text{ DIV } 256), VL); \)
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 read 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.

<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>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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

SVE

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

```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(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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 imm4 0 0 1 Pg Rn Zt

SVE

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

\texttt{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
\texttt{CheckSPAlignment();}
if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
base = SP[];
else
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
base = X[n];
addr = base + offset * 16;
for e = 0 to elements-1
if \texttt{ElemP}[mask, e, esize] == '1' then
\texttt{Elem}[result, e, esize] = Mem[addr, mbytes, AccType_NORMAL];
else
\texttt{Elem}[result, e, esize] = Zeros();
addr = addr + mbytes;
\texttt{Z[t] = Replicate(result, VL DIV 128);}
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 read 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)

**SVE**

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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'
    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 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 read 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.

<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 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>

\begin{array}{c|c|c|c}
\text{imm4} & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}

SVE

LD1RQD \{ <Zt>.D \}, <Pg>/Z, [<Xn|SP>{, #<imm}>]

\[
\text{if } \neg \text{HaveSVE}() \text{ then UNDEFINED;}
\]
\[
\text{integer } t = \text{UInt}(Zt);
\]
\[
\text{integer } n = \text{UInt}(Rn);
\]
\[
\text{integer } g = \text{UInt}(Pg);
\]
\[
\text{integer } esize = 64;
\]
\[
\text{integer } offset = \text{SInt}(imm4);
\]

Assembler Symbols

\begin{itemize}
\item <Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
\item <Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\item <Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
\item <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.
\end{itemize}

Operation

\[
\text{CheckSVEEnabled}();
\]
\[
\text{integer elements} = 128 \text{ DIV } esize;
\]
\[
\text{bits(64) base;}
\]
\[
\text{bits(64) addr;}
\]
\[
\text{bits(PL)} \text{ mask} = \text{P}[g]; \text{ // low 16 bits only}
\]
\[
\text{bits(128) result;}
\]
\[
\text{constant integer mbytes} = \text{esize} \text{ DIV } 8;
\]
\[
\text{if } n == 31 \text{ then}
\]
\[
\text{CheckSPAlignment}();
\]
\[
\text{if } \text{HaveMTEExt}() \text{ then SetNotTagCheckedInstruction(TRUE);}
\]
\[
\text{base} = \text{SP}[];
\]
\[
\text{else}
\]
\[
\text{if } \text{HaveMTEExt}() \text{ then SetNotTagCheckedInstruction(FALSE);}
\]
\[
\text{base} = \text{X}[n];
\]
\[
\text{addr} = \text{base} + \text{offset} \cdot 16;
\]
\[
\text{for } e = 0 \text{ to elements-1}
\]
\[
\text{if } \text{ElemP}[\text{mask, e, esize}] == \text{\'1\'} \text{ then}
\]
\[
\text{Elem}[\text{result, e, esize}] = \text{Mem}[\text{addr, mbytes, AccType_NORMAL}];
\]
\[
\text{else}
\]
\[
\text{Elem}[\text{result, e, esize}] = \text{Zeros}();
\]
\[
\text{addr} = \text{addr} + \text{mbytes};
\]
\[
\text{Z}[t] = \text{Replicate}\text{(result, VL DIV 128)};
\]

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
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 read 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 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0

SVE

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 read 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.

| 31 30 29 28 27 26 25 24 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 | imm4 | 0 0 1 | Pg | Rn | Zt |

SVE

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
integer mbytes = esize DIV 8;
if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 read 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | Rm | 0  | 0  | 0  | Pg | 0  | 0  | 0  | Rn | 0  | 0  | 0  | Zt | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE

LD1RQH { <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;

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 read 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 0 0 0 1 | 0 0 1 | Pg | Rn | Zt |

SVE

LD1RQW { <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 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
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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 scalar base address and scalar index which is multiplied by 4 and added to the base address.

Inactive elements will not read 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.

\[
\begin{array}{cccccccccccccccccccccccccccccc}
\hline
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & Rm & 0 & 0 & 0 & Pg & Rn & Zt
\end{array}
\]

SVE

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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);
**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

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

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

64-bit element

LD1RSB  {  <Zt> },  <Pg>/Z,  [<Xn|SP>{,  #<imm}>]
64-bit element

LD1RSB \{ \langle Z_t \rangle.D, \langle Pg \rangle/Z, [\langle X_n | SP \rangle\{, \#\langle imm \rangle\}] \}

if !HaveSVE() then UNDEFINED;
integer t = UInt(Z_t);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 64;
integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);

Assembler Symbols

<Z_t> 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.
<X_n|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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
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 0 1 | imm6            | 1 0 1            | Pg              |
|                  |                 |                  | Rn              |
|                  |                 |                  | Zt              |

32-bit element

LD1RSH { <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 = 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 0 1 | imm6            | 1 0 0            | Pg              |
|                  |                 |                  | Rn              |
|                  |                 |                  | Zt              |

64-bit element

LD1RSH { <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 = 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

```plaintext
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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 Elem[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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LD1RSW

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.

<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 1 1</td>
</tr>
</tbody>
</table>

SVE

LD1RSW { <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 = 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 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" 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;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**imm6** | **Pg** | **Rn** | **Zt**

### 32-bit element

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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**imm6** | **Pg** | **Rn** | **Zt**

### 64-bit element

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
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 read 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

32-bit element

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

64-bit element

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

```c
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 SetNotTagCheckedInstruction(FALSE);
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 read 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>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>1</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>1</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

LD1SB {<Zt>.H}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

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

LD1SB {<Zt>.S}, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

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

LD1SB (scalar plus immediate)
64-bit element

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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
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 read 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>
<th>Rm</th>
<th>0 1 0</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !\texttt{HaveSVE}() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = \texttt{UInt}(Zt);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
integer g = \texttt{UInt}(Pg);
integer esize = 16;
integer msize = 8;
boolean unsigned = FALSE;

### 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>
<th>Rm</th>
<th>0 1 0</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]

if !\texttt{HaveSVE}() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = \texttt{UInt}(Zt);
integer n = \texttt{UInt}(Rn);
integer m = \texttt{UInt}(Rm);
integer g = \texttt{UInt}(Pg);
integer esize = 32;
integer msize = 8;
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>
<th>Rm</th>
<th>0 1 0</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
64-bit element

LD1SB \{ \text{<Zt>}.D }, \text{<Pg>}/Z, [\text{<Xn|SP>}, \text{<Xm>}]

if \! \text{HaveSVE}() \text{ then UNDEFINED;}
if \text{Rm} == '11111' \text{ then UNDEFINED;}
integer \text{t} = \text{UInt}(\text{Zt});
integer \text{n} = \text{UInt}(\text{Rn});
integer \text{m} = \text{UInt}(\text{Rm});
integer \text{g} = \text{UInt}(\text{Pg});
integer \text{esize} = 64;
integer \text{msize} = 8;
boolean \text{unsigned} = \text{FALSE};

Assembler Symbols

\text{<Zt>} \quad \text{Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.}
\text{<Pg>} \quad \text{Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.}
\text{<Xn|SP>} \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.}
\text{<Xm>} \quad \text{Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.}

Operation

\text{CheckSVEEnabled}();
integer \text{elements} = \text{VL} \text{ DIV esize};
bits(64) \text{base};
bits(64) \text{addr};
bits(PL) \text{mask} = \text{P}[g];
bits(\text{VL}) \text{result};
bits(\text{msize}) \text{data};
bits(64) \text{offset} = \text{X}[m];
constant integer \text{mbytes} = \text{msize} \text{ DIV 8};
if \text{HaveMTEExt()} \text{ then SetNotTagCheckedInstruction(FALSE)};
if \text{n} == 31 then
  \text{CheckSPAlignment}();
  \text{base} = \text{SP}[];
else
  \text{base} = \text{X}[n];
for \text{e} = 0 \text{ to elements-1}
  \text{addr} = \text{base} + \text{UInt}(\text{offset}) \text{ DIV mbytes};
  if \text{ElemP}[\text{mask}, \text{e}, \text{esize}] == '1' \text{ then}
    \text{data} = \text{Mem}[\text{addr}, \text{mbytes}, \text{AccType_NORMAL}];
    \text{Elem}[\text{result}, \text{e}, \text{esize}] = \text{Extend}(\text{data}, \text{esize}, \text{unsigned});
  else
    \text{Elem}[\text{result}, \text{e}, \text{esize}] = \text{Zeros}();
  \text{offset} = \text{offset} + 1;
\text{Z}[t] = \text{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 read 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

```
1 1 0 0 0 1 0 0 0 | Zm 0 0 0 | Pg | Rn | Zt
```

### 32-bit unscaled offset

```
LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]
```

```java
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

```
1 0 0 0 0 1 0 0 0 | Zm 0 0 0 | Pg | Rn | Zt
```

### 32-bit unscaled offset

```
LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]
```

```java
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

```
1 1 0 0 0 1 0 0 0 | Zm 1 0 0 | Pg | Rn | Zt
```

64-bit unscaled offset

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
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 read 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   | 0   | 1   | 0   | 1   | imm5| 1   | 0   | 0   | Pg  | Zn  | Zt  |

32-bit element

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   | 1   | imm5| 1   | 0   | 0   | Pg  | Zn  | Zt  |

64-bit element

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

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 SetNotTagCheckedInstruction(FALSE);

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;
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 read 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

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

64-bit element

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

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
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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;
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 read 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

LD1SH (scalar plus scalar)  Page 1799

32-bit element

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

LD1SH (scalar plus scalar)

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 HaveMTEEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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 0 1 xs 1 Zm 0 0 0 Pg Rn Zt
```

32-bit scaled offset

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 Zm 0 0 0 Pg Rn Zt
```

32-bit unpacked scaled offset

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 Zm 0 0 0 Pg Rn Zt
```

32-bit unpacked unscaled offset

LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]
32-bit unpacked unscaled offset

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  | 0  | 1  | 0  | 0  | 1  | xs | 0  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

32-bit unscaled offset

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  | 0  | 1  | 1  | Zm | 1  | 0  | 0  | Pg | Rn | Zt |

64-bit scaled offset

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  | 0  | 1  | 1  | Zm | 1  | 0  | 0  | Pg | Rn | Zt |
64-bit unscaled offset

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(VL) offset = Z[m];
bits(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 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 read 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 0 1 | imm5 | 1 | 0 | 0 | Pg | Zn | Zt |

SVE

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 SetNotTagCheckedInstruction(FALSE);
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 read 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 0 0 imm4 1 0 1 Pg Rn Zt
```

**SVE**

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
```
**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 read Device memory or signal a fault, and are set to zero in the destination vector.

```
<table>
<thead>
<tr>
<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>0</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);
iteger n = UInt(Rn);
iteger m = UInt(Rm);
iteger g = UInt(Pg);
iteger esize = 64;
iteger 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**

```plaintext
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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 read 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  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

32-bit unpacked unscaled offset

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 1  | 0  | 1  | 0  | 1  | Zm | 0  | 0  | 0  | Pg | Rn | Zt |

32-bit unpacked unscaled offset

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | Pg | Rn | Zt |
64-bit scaled offset

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

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 "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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
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 read 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</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>0</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>imm5</td>
<td>1</td>
<td>1</td>
<td>0</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>

32-bit element

LDIW { <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</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>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>1</td>
<td>imm5</td>
<td>1</td>
<td>1</td>
<td>0</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>

64-bit element

LDIW { <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(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTExe() then SetNotTagCheckedInstruction(FALSE);

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;
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 read 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 diagram]

LD1W { <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 diagram]

LD1W { <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

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
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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 read 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 | Rm | 0 1 0 | Pg | Rn | Zt |

32-bit element

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 0 1 0 0 1 0 | 1 0 1 0 | Rm | 0 1 0 | Pg | Rn | Zt |

64-bit element

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

```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 HaveMTEEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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'
        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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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

```
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;

32-bit unpacked scaled offset

```
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;

32-bit unpacked unscaled offset

```
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;
32-bit unpacked unscaled offset

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

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

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

LD1W (scalar plus vector)
64-bit unscaled offset

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
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 read 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 |
|--------------------------------------------------------|--------------|----------------|-round| 1 0 1 0 0 1 0 0 1 0 | imm4 | 1 1 1 | Pg | Rn | Zt |

SVE

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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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];
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 read 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 = 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

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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];
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 read 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 0 imm4 1 1 1 Pg Rn Zt
```

SVE

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

```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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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)

SVE

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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];
```
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 read 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 0 | imm4 1 1 1 | Pg | Rn | Zt

SVE

LD2H { <Zt1>.H, <Zt2>.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);
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 = \texttt{VL} \text{ DIV esize};
bits(64) base;
bits(64) addr;
bits(PL) mask = \texttt{P}[g];
constant integer mbytes = esize \text{ DIV} 8;
array [0..1] of bits(\texttt{VL}) values;

if \ n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = \texttt{SP}[\ ];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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}[\text{mask}, e, esize] == '1' then
            \texttt{Elem}[\text{values}[r], e, esize] = \texttt{Mem}[addr, mbytes, AccType_NORMAL];
        else
            \texttt{Elem}[\text{values}[r], e, esize] = \texttt{Zeros}();
        addr = addr + mbytes;
    for r = 0 to nreg-1
        \texttt{Z}[(t+r) \ mod 32] = \text{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 read 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 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
Rm Pg Rn Zt
```

SVE

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(VL) mask = P[g];
bits(64) offset = X[m];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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];
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 read 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 0 0 1 0 imm4 1 1 Pg Rn Zt

SVE

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

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
    CheckSPalignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 read 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</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>0</td>
<td>0</td>
<td>1</td>
<td>Rm</td>
<td>1</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>
</tr>
</tbody>
</table>

SVE

LD2W { <Zt1>.S, <Zt2>.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 = 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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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'
            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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

SVE


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 = 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
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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];
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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

<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>0</td>
<td>1</td>
<td>0</td>
<td>Rm</td>
<td>1</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>
</tr>
</tbody>
</table>

SVE

LD3B { <Zt1>.B, <Zt2>.B, <Zt3>.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 = 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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 read 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.
<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 = \text{VL} \div \text{esize};
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = \text{esize} \div 8;
array [0..2] of bits(\text{VL}) values;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    base = X[n];

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{Elem}[values[r], e, esize] = \text{Mem}[addr, mbytes, AccType_NORMAL];
        else
            \text{Elem}[values[r], e, esize] = \text{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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

![Hexadecimal 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 0 1 0 0 1 0 1 1 1 0 1 1 0 | Rm   | 1 1 0 | Pg   | Rn   | Zt   |

SVE

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

```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..2] of bits(VL) values;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

In the 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  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | imm4| 1  | 1  | 1  | Pg | Rn | Zt |

SVE


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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
    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 read 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 0  Rm  1 1 0  Pg  Rn  Zt
```

SVE

LD3H (scalar plus scalar)

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

\[
\text{CheckSVEEnabled}();
\text{integer elements} = \text{VL} \div \text{esize};
\text{bits}(64) \text{ base};
\text{bits}(64) \text{ addr};
\text{bits}(PL) \text{ mask} = P[g];
\text{bits}(64) \text{ offset} = X[m];
\text{constant integer mbytes} = \text{esize} \div 8;
\text{array } [0..2] \text{ of bits(VL) values};
\]

if \text{HaveMTEExt()} then \text{SetNotTagCheckedInstruction}(\text{FALSE});

if \text{n} == 31 then
  \text{CheckSPAlignment}();
  \text{base} = SP[];
else
  \text{base} = X[n];

for \text{e} = 0 to \text{elements}-1
  \text{addr} = \text{base} + \text{UInt}(\text{offset}) \times \text{mbytes};
for \text{r} = 0 to \text{nreg}-1
  \text{if} \ \text{ElemP[mask, e, esize]} == '1' \ \text{then}
    \text{Elem[values[r], e, esize]} = \text{Mem[addr, mbytes, AccType_NORMAL]};
  \text{else}
    \text{Elem[values[r], e, esize]} = Zeros();
  \text{addr} = \text{addr} + \text{mbytes};
  \text{offset} = \text{offset} + \text{nreg};
for \text{r} = 0 to \text{nreg}-1
  \text{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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

\[
\begin{array}{cccccccccccccccccccc}
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & \text{imm4} & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

SVE

\[
\text{LD3W} \{ \text{<Zt1>}.S, \text{<Zt2>}.S, \text{<Zt3>}.S \}, \text{<Pg>}/Z, [\text{<Xn|SP>}, \#<\text{imm}>], \text{MUL VL} \}
\]

\[
\text{if} \ !\text{HaveSVE()} \text{ then UNDEFINED;}
\]
\[
\text{integer } t = \text{UInt}(\text{Zt});
\]
\[
\text{integer } n = \text{UInt}(\text{Rn});
\]
\[
\text{integer } g = \text{UInt}(\text{Pg});
\]
\[
\text{integer } \text{esize} = 32;
\]
\[
\text{integer } \text{offset} = \text{SInt}(\text{imm4});
\]
\[
\text{integer } \text{nreg} = 3;
\]

Assembler Symbols

\[
\begin{align*}
\text{<Zt1>} & \quad \text{Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.} \\
\text{<Zt2>} & \quad \text{Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.} \\
\text{<Zt3>} & \quad \text{Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.} \\
\text{<Pg>} & \quad \text{Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.} \\
\text{<Xn|SP>} & \quad \text{Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.} \\
\text{<imm>} & \quad \text{Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.}
\end{align*}
\]
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
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = `SP`[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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];
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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.

### SVE

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;

### 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 HaveMTExe() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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];
**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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

![Binary Representation](binary_representation.png)

### SVE


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

\textbf{CheckSVEEnabled}();
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(64) base;
bits(64) addr;
bits(\texttt{PL}) mask = \texttt{P}[g];
constant integer mbytes = \texttt{esize} \texttt{DIV} 8;
array [0..3] of bits(\texttt{VL}) values;

if n == 31 then
  \textbf{CheckSPAlignment}();
  if \texttt{HaveMTEExt}() then SetNotTagCheckedInstruction(\texttt{TRUE});
  base = \texttt{SP}[];
else
  if \texttt{HaveMTEExt}() then SetNotTagCheckedInstruction(\texttt{FALSE});
  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];
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 read 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 0 1 1 | Rm | 1 1 0 | Pg | Rn | Zt

SVE


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

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSParallelignment();
    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 read 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 1 1 0 imm4 1 1 1 Pg Rn Zt |

**SVE**


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 nbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP();
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 read 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 1 1 1 Rm 1 1 0 Pg Rn Zt
```

**SVE**

```
LD4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.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 = 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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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];
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 read 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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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];
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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE


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);
ingteger esize = 16;
ingteger 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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

SVE


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;
b Identification understanding
bits(64) base;
b Identification understanding
bits(64) addr;
b Identification understanding
bits(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 read Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.

---

**SVE**

```
```

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;

**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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 read 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 0 0 1</td>
</tr>
</tbody>
</table>

32-bit element

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

<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>

64-bit element

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.
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 SetNotTagCheckedInstruction(FALSE);
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;
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 read 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

8-bit element

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

16-bit element

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

32-bit element

LDFF1B {<Zt>}
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;

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

```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;
bits(64) offset = X[m];
constant integer mbyte = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 then
  CheckSPAlignment();
  base = SP[];
else
  base = X[n];
for e = 0 to elements-1
  if ElemP[mask, e, esize] == '1'
    addr = base + UInt(offset) * mbytes;
    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 & ConstrinUnpredictableBool(Unpredictable_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
      elseif ConstrinUnpredictableBool(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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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 | 1 | Pg | Rn | Zt |

32-bit unpacked unscaled offset

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

| 31 30 29 28 27 26 25 24 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 | 1 | Pg | Rn | Zt |

32-bit unscaled offset

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

| 31 30 29 28 27 26 25 24 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 | 1 | Pg | Rn | Zt |
64-bit unscaled offset

LDFF1B { <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 = 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;
bounds(64) base;
bounds(64) addr;
bounds(VL) offset;
bounds(PL) mask = P[g];
bounds(VL) result;
bounds(VL) orig = Z[t];
bounds(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 then
  CheckSVEAlignment();
  base = SP[];
else
  base = X[n];
  offset = Z[m];
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);
    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;
  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 & 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];
  end;
else
  Elem[result, e, esize] = Extend(data, esize, unsigned);
end;
Z[t] = result;
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 read Device memory or signal faults, and are set to zero in the destination 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>1</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>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>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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

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 \text{ 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 \text{ DIV} 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

for e = 0 to elements-1
if \text{ElemP}[mask, e, esize] == '1' then
  addr = \text{ZeroExtend}(\text{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) = (\text{Zeros}(msize), FALSE);
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    \text{ElemFFR}[e, esize] = '0';
  // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
  unknown = unknown || \text{ElemFFR}[e, esize] == '0';
  if unknown then
    if !fault && \text{ConstrainUnpredictableBool}(Unpredictable_SVELDNFDATA) then
      \text{Elem}[result, e, esize] = \text{Extend}(data, esize, unsigned);
    elseif \text{ConstrainUnpredictableBool}(Unpredictable_SVELDNFZERO) then
      \text{Elem}[result, e, esize] = \text{Zeros}();
    else // merge
      \text{Elem}[result, e, esize] = \text{Elem}[orig, e, esize];
    else
      \text{Elem}[result, e, esize] = \text{Extend}(data, esize, unsigned);
  end
Z[t] = result;
**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 read 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 1 1 1 1 | Rm | 0 1 1 | Pg | Rn | Zt |

**SVE**

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 & ConstrinUnpredictable Bool(Unpredictable_SVELDNFDATA) then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
      elsif ConstrinUnpredictable Bool(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;
**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 read 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</td>
</tr>
</tbody>
</table>

### 32-bit unpacked unscaled offset

LDFF1D {<Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]

```java
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;
```

### 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</td>
</tr>
</tbody>
</table>
64-bit scaled offset

LDFF1D { <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

LDFF1D { <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 "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(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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 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];
        end
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    end
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 read 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

![Binary representation of 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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</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>
<td></td>
</tr>
</tbody>
</table>
```

```
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

![Binary representation of 64-bit element]

```
| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | Pg | Zn | Zt |
```

```
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 SetNotTagCheckedInstruction(FALSE);
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 && ConstrinUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrinUnpredictableBool(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 read 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>
<th>Rm</th>
<th>0 1 1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
</table>

16-bit element

LDFF1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

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

<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>Rm</th>
<th>0 1 1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
</table>

32-bit element

LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

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

<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>Rm</th>
<th>0 1 1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
</table>

64-bit element
64-bit element

LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

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

```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;
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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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];
        end
        (data, fault) = (Zeros(msize), FALSE);
    end
    // FFR elements set to FALSE following a suppressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';
    end
    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault &
            ConstrainUnpredictableBool(Unpredictable_SVLDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif
            ConstrainUnpredictableBool(Unpredictable_SVLDNFDZERO) then
            Elem[result, e, esize] = Zeros();
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
        end
    end
    Elem[result, e, esize] = Extend(data, esize, unsigned);
    offset = offset + 1;
Z[t] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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

```
LDFF1H { <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

```
LDFF1H { <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

```
LDFF1H { <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 unscaled offset

LDFF1H { <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

LDFF1H { <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

LDFF1H { <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

LDFF1H (scalar plus vector)
64-bit unscaled offset

LDFF1H { <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

```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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 & 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];
      else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
      Elem[result, e, esize] = Elem[orig, e, esize];
  else
    Elem[result, e, esize] = Extend(data, esize, unsigned);
Z[t] = result;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 read 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  | 0  | 1  | Pg | Zn | Zt |

**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.

### 64-bit element

| 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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | imm5| 1  | 0  | 1  | Pg | Zn | Zt |

**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.

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);
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 SetNotTagCheckedInstruction(FALSE);
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;
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 read 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

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

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

LDFF1SB {
  <Zt> }, <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;
64-bit element

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

```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;
bits(64) offset = X[m];
constant integer mbyte = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 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 && ConstrinUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrinUnpredictableBool(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;
```
**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 read 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 | 1 | Pg | Rn | Zt
```

### 32-bit unscaled offset

```
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

```
31 30 29 28 27 26 25 24 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 | 1 | Pg | Rn | Zt
```

### 32-bit unscaled offset

```
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

```
31 30 29 28 27 26 25 24 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 1 0 0 1 0 0 | Zm | 1 | 0 | 1 | Pg | Rn | Zt
```
64-bit unscaled offset

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>

LDFF1SB (scalar plus vector) Page 1899
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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 & ConstrUnpredictableBool(Unpredictable_SVELDNFDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrUnpredictableBool(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 (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 read 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

|   | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 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  |

**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

|   | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**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 SetNotTagCheckedInstruction(FALSE);
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;
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 read 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

```
  Rm  0 1 1  |  Pg  |  Rn  |  Zt
```

### 32-bit element

LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

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

```
  Rm  0 1 1  |  Pg  |  Rn  |  Zt
```

### 64-bit element

LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

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];
bits(msize) data;
bits(64) offset = X[m];
constant integer mbyte = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 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 && ConstrinUnpredictableBool(Unpredictable_SVELDNFDATA) then
                Elem[result, e, esize] = Extend(data, esize, unsigned);
            elsif ConstrinUnpredictableBool(Unpredictable_SVELDNFZERO) then
                Elem[result, e, esize] = Zeros();
            else
                Elem[result, e, esize] = Elem[orig, e, esize];
            end
            offset = offset + 1;
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    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 read 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

LDFF1SH (scalar plus vector)

32-bit scaled offset

LDFF1SH {<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

LDFF1SH {<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

LDFF1SH {<Zt>.O}, <Pg>/Z, [<Xn|SP>, <Zm>.O, <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 unscaled offset

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></td>
<td></td>
<td></td>
<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 unscaled offset

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>0</td>
<td>1</td>
<td>0</td>
<td>0</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></td>
<td></td>
<td></td>
<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>

64-bit scaled offset

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>0</td>
<td>1</td>
<td>0</td>
<td>0</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></td>
<td></td>
<td></td>
<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>
64-bit unscaled offset

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(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 SetNotTagCheckedInstruction(FALSE);
if n == 31 then
  CheckSPAlignment();
  base = SP[];
else
  base = X[n];
  offset = Z[m];
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);
    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;
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 read Device memory or signal faults, and are set to zero in the destination 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>1</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>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>

SVE

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

```plaintext
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 SetNotTagCheckedInstruction(FALSE);
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_SVELDNPDATA) then
      Elem[result, e, esize] = Extend(data, esize, unsigned);
    elsif ConstrainUnpredictableBool(Unpredictable_SVELDNPZERO) 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0

Rm | Pg | Rn | Zt

SVE

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 mbyte = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 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);
    else
        // 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);
        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 read 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</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>0</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>

**LDFF1SW** {<Zt>.D }, <Pg>/Z, [<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 unsigned = FALSE;
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>0</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>

**LDFF1SW** {<Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```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 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>1</td>
<td>0</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></td>
<td></td>
<td></td>
<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>

LDFF1SW (scalar plus vector) Page 1913
64-bit scaled offset

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

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 then
    CheckSPAlignment();
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
    end
    else
        (data, fault) = (Zeros(msize), FALSE);
        // FFR elements set to FALSE following a supressed access/fault
        faulted = faulted || fault;
    end
    if faulted then
        ElemFFR[e, esize] = '0';
    end
    // 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];
        end
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    end
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 read 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  | 1  | 0  | 0  | 1  | imm5 | 1  | 1  | 1  | Pg  | Zn  | Zt  |

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | imm5 | 1  | 1  | 1  | Pg  | Zn  | Zt  |

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 SetNotTagCheckedInstruction(FALSE);
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);
  end
  // FFR elements set to FALSE following a supressed access/fault
  faulted = faulted || fault;
  if faulted then
    ElemFFR[e, esize] = '0';
  end
  // 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];
    end
  end
  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 read 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 |
```

LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

```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 = 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 1 |
```

LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

```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 = 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

```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;
bits(64) offset = X[m];
constant integer mbyte = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulked = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
```
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 read 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

```plaintext
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

```plaintext
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

```plaintext
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 unscaled offset

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

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

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

LDFF1W (scalar plus vector)
64-bit unscaled offset

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;
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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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
    (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';
  end

  // 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];
    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 read 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  | 1  | 0  | 1  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

8-bit element

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |

16-bit element

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | imm4| 1  | 0  | 1  | Pg | Rn | Zt |
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);

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.
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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
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 read 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 1 1 1 1 1 1      | imm4 1 0 1                      | Pg                              |
| Rn                             | Zt                              |

SVE

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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 && 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;
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 read 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 0 1 0 1 0 1 1 1 0 0 0 0 1 0 0 1 0 1 1 imm4</td>
</tr>
</tbody>
</table>

16-bit element

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 0 1 0 1 1 0 1 imm4</td>
</tr>
</tbody>
</table>

32-bit element

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 0 1 0 0 1 1 1 1 imm4</td>
</tr>
</tbody>
</table>
64-bit element

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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 && 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
    addr = addr + mbytes;
Z[t] = result;
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 read 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 1 1 0 1</td>
</tr>
</tbody>
</table>

16-bit element

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 1 0 1 1</td>
</tr>
</tbody>
</table>

32-bit element

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 1 0 0 1</td>
</tr>
</tbody>
</table>
64-bit element

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
   CheckSPAlignment();
   if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
      base = SP[];
else
   if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
      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 && 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
      Elem[result, e, esize] = Extend(data, esize, unsigned);
   end
   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 read 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  | 1  | imm4 | 1  | 0  | 1  | Pg | Rn | Zt |

32-bit element

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | imm4 | 1  | 0  | 1  | Pg | Rn | Zt |

64-bit element

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 & ConstrlnUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elseif ConstrlnUnpredictableBool(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;
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 read 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 |  0 |  1 |  1 |  imm4 |  1 |  0 |  1 | Pg | Rn | Zt |

SVE

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

```plaintext
CheckSVEEnabled();
integer elements = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
basis(VL) result;
basis(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
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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 && 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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
LDNF1W

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 read 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 | imm4 | 1 | 0 | 1 | Pg | Rn | Zt |

32-bit element

```
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

```
31 30 29 28 27 26 25 24 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 | 1 | imm4 | 1 | 0 | 1 | Pg | Rn | Zt |

64-bit element

```
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

CheckSVEEnabled();
integer elements = \texttt{VL} \text{ DIV} \texttt{esize};
bits(64) base;
bits(64) addr;
bits(PL) mask = \texttt{P}[g];
bits(VL) result;
bits(VL) orig = \texttt{Z}[t];
bits(msize) data;
constant integer mbytes = msize \text{ DIV} 8;
boolean fault = \texttt{FALSE};
boolean faulted = \texttt{FALSE};
boolean unknown = \texttt{FALSE};
if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = \texttt{SP}[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    base = \texttt{X}[n];
addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if \texttt{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
        \texttt{ElemFFR}[e, esize] = '0';
    // Value becomes CONstrained UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || \texttt{ElemFFR}[e, esize] == '0';
    if unknown then
        if !fault && \texttt{ConstrainUnpredictableBool(Unpredictable\_SVELDNFDATA)} then
            \texttt{Elem}[result, e, esize] = \texttt{Extend}(data, esize, unsigned);
        elsif \texttt{ConstrainUnpredictableBool(Unpredictable\_SVELDNFZERO)} then
            \texttt{Elem}[result, e, esize] = \texttt{Zeros}();
        else // merge
            \texttt{Elem}[result, e, esize] = \texttt{Elem}[orig, e, esize];
        end
        \texttt{Elem}[result, e, esize] = \texttt{Extend}(data, esize, unsigned);
    end
    addr = addr + mbytes;
\texttt{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 read 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 0 1 0 | 0 0 0 0 | Rm  | 1 0 1 | Pg  | Zn  | Zt
```

32-bit unscaled offset

```
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 0 | Rm  | 1 1 0 | Pg  | Zn  | Zt
```

64-bit unscaled offset

```
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

\textbf{CheckSVEEnabled}();
integer elements = \textbf{VL} \text{ DIV} \text{ esize};
bits(\textbf{VL}) base = \textbf{Z}[n];
bits(64) offset = \textbf{X}[m];
bits(64) addr;
bits(PL) mask = \textbf{P}[g];
bits(\textbf{VL}) result;
bits(msize) data;
constant integer mbytes = msize \text{ DIV} 8;

\text{if HaveMTEExt}() \text{ then SetNotTagCheckedInstruction}(FALSE);

\text{for e = 0 to elements-1}
\text{if} \quad \text{ElemP}[\text{mask, e, esize}] == '1' \text{ then}
\quad \text{addr} = \text{ZeroExtend(Elem[base, e, esize], 64) + offset;}
\quad \text{data} = \text{Mem[addr, mbytes, AccType_STREAM]};
\quad \text{Elem[result, e, esize]} = \text{Extend(data, esize, unsigned)};
\text{else}
\quad \text{Elem[result, e, esize]} = \text{Zeros}();
\text{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 read 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 imm4   1 1 1                  Pg       Rn       Zt
```

SVE

```
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;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
    CheckSPAAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[ ];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
```
LDNT1B (scalar plus scalar)

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 read 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.

SVE

LDNT1B { <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 = 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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 read 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.

### SVE2

```plaintext
LDNT1D { <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 = 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.
- `<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 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 SetNotTagCheckedInstruction(FALSE);
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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.

SVE

LDNT1D { <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 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;
constant integer mbytes = esize DIV 8;
if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 read 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 1 0 1 1 0
Rm 1 1 0
Pg
Rn
Zt

SVE

LDNT1D { <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 = 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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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 (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 read 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

\[
\begin{array}{cccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & | & Rm & 1 & 0 & 1 & | & Pg & | & Zn & | & Zt
\end{array}
\]

### 32-bit unscaled offset

```
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

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & | & Rm & 1 & 1 & 0 & | & Pg & | & Zn & | & Zt
\end{array}
\]

### 64-bit unscaled offset

```
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();
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 HaveMTExeExt() then SetNotTagCheckedInstruction(FALSE);

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 of 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 read 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  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

SVE

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(PL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
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 read 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.

SVE

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;

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(VL) mask = P[g];
bits(VL) result;
constant integer mbytes = esize DIV 8;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 then
    CheckSPAlignment();
    base = SP[ ];
else
    base = X[n];
offset = X[m];
for e = 0 to elements-1
    addr = base + UInt(offset) * mbytes;
    if Elem[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;
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 read 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
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

```
31 30 29 28 27 26 25 24 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
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 SetNotTagCheckedInstruction(FALSE);
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 read 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

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

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();
integer elements = \texttt{VL} \div \texttt{esize};
bits(\texttt{VL}) \texttt{base} = \texttt{Z[n]};
bits(64) \texttt{offset} = \texttt{X[m]};
bits(64) \texttt{addr};
bits(PL) \texttt{mask} = \texttt{P[g]};
bits(\texttt{VL}) \texttt{result};
bits(msize) \texttt{data};
constant integer mbytes = msize \div 8;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

for \texttt{e} = 0 to \texttt{elements-1}
    if \texttt{ElemP[mask, e, esize]} == '1' then
        \texttt{addr} = \texttt{ZeroExtend(Elem[base, e, esize], 64)} + \texttt{offset};
        \texttt{data} = \texttt{Mem[addr, mbytes, AccType_STREAM]};
        \texttt{Elem[result, e, esize]} = \texttt{Extend(data, esize, unsigned)};
    else
        \texttt{Elem[result, e, esize]} = \texttt{Zeros()};

\texttt{Z[t]} = \texttt{result};

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 read 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.

SVE2

LDNT1SW { <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 = 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 HaveMTEEExt() then SetNotTagCheckedInstruction(FALSE);
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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 read 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>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

32-bit unscaled offset

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 30 29 28 27 26 25 24 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 0 1 0 0 0 1 0 1 0 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>

64-bit unscaled offset

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;
b研制(PL) mask = P[g];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;
if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
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;
```
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 read 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 1 1 1 | Pg | Rn | Zt

SVE

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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
LDNT1W (scalar plus scalar)

Contiguous load non-temporal words to vector (scalar 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 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 read 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

<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>
<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>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>Rm</td>
<td>1</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>
</tr>
</tbody>
</table>

SVE

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();
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 SetNotTagCheckedInstruction(FALSE);
if n == 31 then
  CheckSPAlignment();
  base = SP[];
else
  base = X[n];
offset = X[m];
for e = 0 to elements-1
  addr = base + UInt(offset) * mbytes;
  ifElemP[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.

<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 1 0</td>
</tr>
</tbody>
</table>

SVE

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 SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
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.

### SVE

```
LDR <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(64) base;
integer offset = imm * elements;
bits(VL) result;
if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 |
|-----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|               | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 1   | 1   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   |

SVE

LSL <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 = 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>0x</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
   bits(esize) element1 = Elem[operand1, e, 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE

LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

if !HaveSVE() then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
int g = UInt(Pg);
int dn = UInt(Zdn);
int 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.

• The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.

• A predicated MOVPRFX instruction using the same governing predicate register and destination element size as
  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  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | Pg | Zm | Zdn|
```

SVE

\[ \text{LSL } <Zdn>,.<T>, <Pg>/M, <Zdn>,.<T>, <Zm>.<T> \]

if \( \text{HaveSVE}() \) then UNDEFINED;

integer esize = 8 \( \ll \) UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);

Asmbleer 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

\( \text{CheckSVEEnabled}() ; \)
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
  \( \text{bits(esize) element1} = \text{Elem}[\text{operand1}, e, \text{esize}] ; \)
  \( \text{bits(esize) element2} = \text{Elem}[\text{operand2}, e, \text{esize}] ; \)
  integer shift = Min(UInt(element2), esize);
  if \( \text{ElemP[mask, e, esize]} = 1 \) then
    \( \text{Elem[result, e, esize]} = \text{LSL(element1, shift)} ; \)
  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 that conforms to all of the
following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this
  instruction.
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.

<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>

SVE

```
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  |

**SVE**

**LSL <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] = 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.
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  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | Pg | Zm | Zdn |

SVE

LSLR <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 ElemP[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | Pg | tszl| imm3| Zdn|

SVE

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);
dnt = 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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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>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>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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

LSR <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] = 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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and destination element size as 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 0  | 0  | P  | g  | Z  | m  | Z  | d  | n  |

SVE

LSR <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] = 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 that conforms to all of the
following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>tsh</td>
</tr>
<tr>
<td>26</td>
<td>tsz</td>
</tr>
<tr>
<td>25</td>
<td>imm</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>Zn</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>

SVE

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(tsize:imm3);

Assemble 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.
<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) 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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 30 29 28 27 26 25 24 23 22 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 0 | size | 1 | Zm | 1 0 0 0 0 0 1 | Zn | Zd |

SVE

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  |

SVE

LSRR <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 ElemP[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
MAD

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.

| 31 30 29 28 27 26 25 24 23 22 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 | Zm |        | Pg   | Za |        | Zdn |        |

SVE

MAD \(<Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>\)

if !HaveSVE() then UNDEFINED;
integer esize = 8 \( \ll \) UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = 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.
\(<Za>\) 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

### SVE2

MATCH <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] = '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';

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.
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.

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | Pg | Zn | Zda |

SVE

MLA \(<Z_{da}>, <T>\), \(<Pg>/M, <Zn>.<T>, <Zm>.<T>\)

if \(!\text{HaveSVE}()\) then UNDEFINED;
integer esize = 8 \ll\ \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 = 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

\(\text{CheckSVEEnabled}();\)
integer elements = \text{VL} \div \text{esize};
bits(PL) mask = \text{P}[g];
bits(VL) operand1 = \text{Z}[n];
bits(VL) operand2 = \text{Z}[m];
bits(VL) operand3 = \text{Z}[da];
bits(VL) result;
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}]);
if \text{ElemP}[\text{mask}, e, \text{esize}] == '1'
then
integer product = element1 * element2;
if sub_op
\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand3}, e, \text{esize}] - product;
else
\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand3}, e, \text{esize}] + product;
else
\text{Elem}[\text{result}, e, \text{esize}] = \text{Elem}[\text{operand3}, e, \text{esize}];
\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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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


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

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


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) 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]);
  integer element3 = UInt(Elem[result, e, esize]);
  integer res = element3 + element1 * element2;
  Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

  • An unpredicated MOVPRFX instruction.
**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  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 1  |

**SVE**

MLS \(<Zda>, <T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>\)

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = 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**

```c
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
    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[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
**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
```

**16-bit**

```mls
```

```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

```
31 30 29 28 27 26 25 24 23 22 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 |i2| Zm 0 0 0 0 1 1 | Zn | Zda
```

**32-bit**

```mls
MLS <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

```
31 30 29 28 27 26 25 24 23 22 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 |i1| Zm 0 0 0 0 1 1 | Zn | Zda
```

**64-bit**

```mls
```

```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

CheckSVEEnabled();

integer elements = \text{VL} \div \text{esize};
integer eltspersegment = 128 \div \text{esize};
bits(\text{VL}) \text{operand1} = Z[n];
bits(\text{VL}) \text{operand2} = Z[m];
bits(\text{VL}) \text{result} = Z[da];

for e = 0 to elements-1
    integer segmentbase = e - e \mod eltspersegment;
    integer s = segmentbase + index;
    integer \text{element1} = \text{UInt}(\text{Elem}[^{\text{operand1}}, e, \text{esize}]);
    integer \text{element2} = \text{UInt}(\text{Elem}[^{\text{operand2}}, s, \text{esize}]);
    integer \text{element3} = \text{UInt}(\text{Elem}[^{\text{result}}, e, \text{esize}]);
    integer \text{res} = \text{element3} - \text{element1} \times \text{element2};
    \text{Elem}[^{\text{result}}, e, \text{esize}] = \text{res}<\text{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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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.

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.

```
31 30 29 28 27 26 25 24 23 22 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 0 0 | M 0 0 | Pg  | Zn  |  Zd
SVE
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.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<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>

<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 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(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.

Internally, the MOVPRFX instruction uses the `Zn` and `Zd` fields to encode the source and destination vector registers, respectively.

SVE

MOVPRFX <Zd>, <Zn>

- if `!HaveSVE()` then 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.
- `<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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
Multiply-subtract vectors (predicated), writing multiplicand \([Zdn = Za - Zdn \cdot Zm]\).

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.

| 31 30 29 28 27 26 25 24 23 22 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 | 1 1 1 | Pg | Za | Zdn |

SVE

```c
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**

```c
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX instruction.
  • A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

![Register Table]

<table>
<thead>
<tr>
<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>
</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>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>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>B</td>
<td>01</td>
<td>H</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 30 29 28 27 26 25 24 23 22 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 | size | 1  1  0  |  0  0  |  0  1  1  0 | imm8 | Zdn |

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX 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.

---

SVE2

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 | i3h | i3l | Zm | 1 1 1 1 1 1 0 | Zn | Zd
```

16-bit

```

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 0 1 0 1 | i2 | Zm | 1 1 1 1 1 1 0 | Zn | Zd
```

32-bit

```
MUL <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

```
 0 1 0 0 0 1 0 1 1 | i1 | Zm | 1 1 1 1 1 1 0 | Zn | Zd
```

64-bit

```
MUL <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

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

\[
\text{CheckSVEEnabled}();
\]

\[
\text{integer elements} = \text{VL} \div \text{esize};
\]

\[
\text{integer eltspersegment} = 128 \div \text{esize};
\]

\[
\text{bits(} \text{VL} \text{) operand1} = Z[n];
\]

\[
\text{bits(} \text{VL} \text{) operand2} = Z[m];
\]

\[
\text{bits(} \text{VL} \text{) result};
\]

\[
\text{for } e = 0 \text{ to elements-1}
\]

\[
\begin{align*}
\text{integer segmentbase} & = e - e \text{ MOD eltspersegment}; \\
\text{integer } s & = \text{segmentbase + index}; \\
\text{integer element1} & = \text{UInt(} \text{Elem}[\text{operand1, } e, \text{ esize}]); \\
\text{integer element2} & = \text{UInt(} \text{Elem}[\text{operand2, } s, \text{ esize}]); \\
\text{integer res} & = \text{element1 * element2}; \\
\text{Elem}[\text{result, } e, \text{ esize}] & = \text{res}<\text{esize}-1:0>;
\end{align*}
\]

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

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.
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 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | Pm | 0 | 1 | Pg | 1 | Pn | 1 | Pd |

Not setting the condition flags


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 | Pm | 0 | 1 | Pg | 1 | Pn | 1 | Pd |

Setting the condition flags


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

```assembly
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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
NBSL

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  |

SVE2


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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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</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>1</td>
<td>0</td>
<td>1</td>
<td>P</td>
<td>Z</td>
<td>n</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>
<td></td>
</tr>
</tbody>
</table>

SVE

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>&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 = -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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
**NMATCH**

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 | Pd |

**SVE2**

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**

```plaintext
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'
    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';
  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.
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

| 31 30 29 28 27 26 25 24 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 | 0 | Pd |
---|---|---|---|---|---|---|---|

Not setting the condition flags


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 | Pm | 0 1 | Pg | 1 | Pn | 0 | Pd |
---|---|---|---|---|---|---|---|

Setting the condition flags


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] = 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 (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>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></td>
<td></td>
<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>

NOT <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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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

<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 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Not setting the condition flags


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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Setting the condition flags


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 OR (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.
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$S$, and MOV$$. 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  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  |


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  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |


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$S$</td>
<td>$S == '1' &amp;&amp; Pn == Pm &amp;&amp; Pm == Pg$</td>
</tr>
</tbody>
</table>
Alias | Is preferred when
---|---
MOV | S == '0' && Pn == Pm && Pm == Pg

### 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.

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

**SVE**

ORR `<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**

```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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 1 0 1 0 0 0 0 0 0 0 0 | imm13            |
| Zdn                          |
```

SVE

```
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>0xxxxx</td>
<td>S</td>
</tr>
<tr>
<td>0</td>
<td>10xxxxx</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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
The MOVPRFX instruction must specify the same destination register as this instruction.
The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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).

| 31 30 29 28 27 26 25 24 23 22 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 1 | Zm | 0 0 1 1 0 0 | Zn | Zd |

SVE

ORR <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.

Alias Conditions

<table>
<thead>
<tr>
<th>Alias</th>
<th>Is preferred when</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (vector, unpredicated)</td>
<td>Zn == Zm</td>
</tr>
</tbody>
</table>

Operation

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.

<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</td>
</tr>
</tbody>
</table>

SVE

ORV <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 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.
PFALSE

Set all predicate elements to false.
Set all elements in the destination predicate to false.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Pd |

SVE

PFALSE <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();
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.

```
31 30 29 28 27 26 25 24 23 22 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 1 0 0 0 1 1 0 0 0 0 0 | Pg | 0 | Pdn
```

SVE

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;
```

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

<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 0 1</td>
</tr>
</tbody>
</table>

SVE2

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.

```
<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>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>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>
</tr>
</tbody>
</table>

SVE2

PMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' && !HaveSVE2PMULL128() then UNDEFINED;
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

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 + 0, esize DIV 2];
  bits(esize DIV 2) element2 = Elem[operand2, 2*e + 0, 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.
**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 AA64ZFRO_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: 0 | Zm: 0 1 1 | Zn: 1 1 | Zd: 2 2 |

**SVE2**

PMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' && !HaveSVE2PMULL128() then UNDEFINED;
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**

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.
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 FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

### SVE

PNEXT <Pdn>.<T>, <Pg>, <Pdn>.<T>

if !HaveSVE() then UNDEFINED;
integer esize = 8 << UInt(size);
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.
- **<T>** Is the 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.

### Operation

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;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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>

32-bit element

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`;
integer scale = 0;
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>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>

64-bit element

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`;
integer scale = 0;
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>011x</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

```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' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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
```

SVE

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

<table>
<thead>
<tr>
<th>&lt;prfop&gt;</th>
<th>Is the prefetch operation specifier, encoded in “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>#&lt;imm4&gt;</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);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
1 0 0 0 0 1 0 0 0 0 Rm 1 1 0 Pg Rn 0 prfop
```

SVE

PRFB <prfop>, <Pg>, [<Xn|SP>, <Xm>]

if !HaveSVE() then UNDEFINED;
if Rm == '1111' then UNDEFINED;
integer esize = 8;
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 = 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.

- `<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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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 |  0 |  0 |  1 |  Zm|  0 |  0 |  0 |  Pg|  Rn|  0 |  prfop|
```

### 32-bit scaled offset

```
PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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|  0 |  0 |  0 |  Pg|  Rn|  0 |  prfop|
```

### 32-bit unpacked scaled offset

```
PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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 |  0 |  0 |  Pg|  Rn|  0 |  prfop|
```

PRFB (scalar plus vector)
64-bit scaled offset

PRFB <prfop>, <Pg>, [<Xn|SP>], <Zm>.D

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

<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 1 0 0</td>
</tr>
</tbody>
</table>

32-bit element

PRFD <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 = 3;
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 1 0 0</td>
</tr>
</tbody>
</table>

64-bit element

PRFD <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 = 3;
integer offset = UInt(imm5);

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”: 

PRFD (vector plus immediate)  Page 2058
<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 8 in the range 0 to 248, 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' then
        addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

<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>imm6</th>
<th>0 1 1</th>
<th>Pg</th>
<th>Rn</th>
<th>0</th>
<th>prfop</th>
</tr>
</thead>
</table>

SVE

PRFD <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
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 = 3;
integer offset = SInt(imm6);

Assembler Symbols

<table>
<thead>
<tr>
<th>&lt;prfop&gt;</th>
<th>Is the prefetch operation specifier, encoded in “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 |   Rm |
|   1 1 0 |   Pg |   Rn |
|   0 |   prfop |

**SVE**

```plaintext
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_hints = 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>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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**PRFD (scalar plus vector)**

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

![32-bit scaled offset](image)

```plaintext
PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #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

![32-bit unpacked scaled offset](image)

```plaintext
PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #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

![64-bit scaled offset](image)

```plaintext
PRFD <prfop>, <Pg>, <Rn> 0 prfop
```

---

**PRFD (scalar plus vector)**
64-bit scaled offset

PRFD <prfop>, <Pg>, [<Xn|SP>], <Zm>.D, LSL #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 = 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>#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;
bv64 addr;
bv64 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  | 1  | 0  | 0  | 1  | 0  | 0  | | imm5 | 1 | 1 | | | | | | | | Pg | | Zn | 0 | | prfop |

32-bit element

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 | | | | | | | | Pg | | Zn | 0 | | prfop |

64-bit element

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”:  

---

Page 2067
<table>
<thead>
<tr>
<th>prfop</th>
<th>Description</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>

**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' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
    Hint_Prefetch(addr, pref_hint, level, stream);
```

<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.
**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 1 imm6 0 0 1 Pg Rn 0 prfop
```

**SVE**

PRFH `<prfop>`, `<Pg>`, `<Xn|SP>{, #<imm>, MUL VL}`

```
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><code>prfop</code></th>
<th>Operation</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);
**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  | 0  | 1  | 0  | 0  | Rm | 1  | 1  | 0  | Pg | Rn | 0  | prfop |

**SVE**

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

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

<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>

32-bit scaled offset

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');
in_integer scale = 1;

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>

32-bit unpacked scaled offset

PRFH <prfop>, <Pg>, [<Xn|SP>], <Zm>.D, <mod> #1]

if !HaveSVE() then UNDEFINED;
integer esize = 64;
in_integer g = UInt(Pg);
in_integer n = UInt(Rn);
in_integer m = UInt(Zm);
in_integer level = UInt(prfop<2:1>);
in_integer stream = (prfop<0> == '1');
in_integer pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
in_integer offs_size = 32;
in_integer offs_unsigned = (xs == '0');
in_integer scale = 1;

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</td>
</tr>
</tbody>
</table>
64-bit scaled offset

PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #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 = 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>1001</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 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>
</tr>
</tbody>
</table>

32-bit element

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 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>

64-bit element

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”:
<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>111x</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 4 in the range 0 to 124, 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 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | imm6 | 0 | 1 | 0 | Pg | Rn | 0 | prfop |

**SVE**

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.
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);
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
1 0 0 0 0 1 0 1 0 0 0 Rm 1 1 0 Pg Rn 0 prfop
```

SVE

```
PRFW <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]
```

```java
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 = \texttt{VL} \texttt{DIV} \texttt{esize};

\texttt{bits(PL)} mask = \texttt{P}[g];
\texttt{bits(64)} base;
\texttt{bits(64)} offset = \texttt{X}[m];
\texttt{bits(64)} addr;

if \(n == 31\) then
  base = \texttt{SP}[];
else
  base = \texttt{X}[n];

for \(e = 0\) to \(\text{elements-1}\)
  if \texttt{ElemP[mask, e, esize]} == '1'
    addr = base + (\texttt{UInt}(offset) \ll scale);
    \texttt{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 1 0 0 0 | xs | 1 | Zm | 0 | 1 | 0 | Pg | Rn | 0 | prfop |

### 32-bit scaled offset

PRFW `<prfop>`, `<Pg>`, `<Xn|SP>`, `<Zm>.S`, `<mod> #2`

```plaintext
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 |

### 32-bit unpacked scaled offset

PRFW `<prfop>`, `<Pg>`, `<Xn|SP>`, `<Zm>.D`, `<mod> #2`

```plaintext
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 |
64-bit scaled offset

PRFW <prfop>, <Pg>, [<Xn|SP>], <Zm>.D, LSL #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 = 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);
PTEST

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.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 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 | 1 |   |   |   |   |   |   |   |   |   |   |   |

SVE

PTEST <Pg>, <Pn>.

if !HaveSVE() then UNDEFINED;
integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);

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(PL) mask = P[g];
bits(PL) 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

```
Not setting the condition flags

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

```
Setting the condition flags

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

```
Asmber 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>#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>

**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.
PUNPKHI, PUNPKLO

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]
[0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1 0 1 0 0 0 0 0 | Pn 0 | Pd]
```

**PUNPKHI <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]
[0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 1 0 1 0 0 0 0 0 | Pn 0 | Pd]
```

**PUNPKLO <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

```
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.

<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>1</td>
<td></td>
<td>size</td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE2

RADDHNB <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>0</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 halfsize = esize DIV 2;
integer round_const = 1 << (halfsize - 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) >> halfsize;
  Elem[result, 2*e + 0, halfsize] = res<halfsize-1:0>;
  Elem[result, 2*e + 1, halfsize] = 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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SVE2

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.

<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();
type integer elements = VL DIV esize;
bits(VL) operand1 = Z[n];
bits(VL) operand2 = Z[m];
bits(VL) result = Z[d];
type integer halfesize = esize DIV 2;
type 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Zm | 1  | 1  | 1  | 1  | 0  | 0  | 1  | Zn | 1  | 1  | Zd |

SVE2

RAX1 \(<Zd>.D, <Zn>.D, <Zm>.D\)

if !HaveSVE2SHA3() 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 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   |  1   |  0   |  1   |  1   |  1   |  0   |  0   |  P g |  Z n |  Z d |

SVE

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 ElemP[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
**RDFFR (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  | 0  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  |

**SVE**

RDFFR <Pd>.B

```plaintext
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

<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>1</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>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>
</tr>
<tr>
<td>Pg</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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not setting the condition flags

RDFFR <Pd>.B, <Pg>/Z

if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = FALSE;

Setting the condition flags

<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>1</td>
<td>0</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>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>
</tr>
<tr>
<td>Pg</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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Setting the condition flags

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  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | imm6 | Rd |

SVE

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

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.

```
| 31 30 29 28 27 26 25 24 23 22 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   1   0   0   0   0   0   0   0   Pn   0   Pd |
```

**SVE**

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><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>

- `<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.

| 31 30 29 28 27 26 25 24 23 22 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 1 0 0 0 0 1 1 1 0 Zn | Zd |

SVE

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 |

Byte

REBV <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 |

Halfword

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 |

Word

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”:

<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<T>”:

<table>
<thead>
<tr>
<th>size&lt;T&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
  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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 0 1 0 1 0 tszh | tszl | imm3 | 0 0 0 1 1 0 | Zn | Zd |

SVE2

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(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":

tszh tszl <T>
0 00 RESERVED
0 01 B
0 1x H
1 xx S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in "tszh:tszl":

tszh tszl <Tb>
0 00 RESERVED
0 01 H
0 1x S
1 xx D

<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*sizeof) element = Elem[operand, e, 2*sizeof];
  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.
RSHRNT

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.

<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>tszh</td>
<td>1</td>
<td>tszl</td>
<td>imm3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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>
</tr>
</tbody>
</table>

SVE2

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(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></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>
</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></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>
</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  | 1  | 0  | 1  | Zm | 0  | 1  | 1  | 1  | 0  | Zn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SVE2

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 halfsize = esize DIV 2;
integer round_const = 1 << (halfsize - 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) >> halfsize;
  Elem[result, 2*e + 0, halfsize] = res<halfsize-1:0>;
  Elem[result, 2*e + 1, halfsize] = 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 0 1 0 1 | size | 1 |
  Zm 0 1 1 1 1 1 1 | Zn | Zd
```

SVE2

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 halfsize = esize DIV 2;
integer round_const = 1 << (halfsize - 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) >> 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.
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.

```
| 31 30 29 28 27 26 25 24 23 22 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            | Zm              | 1 1 1 1 1 1 0   | Zn              | Zda             |
```

**SVE2**

SABA `<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);
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>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**

```
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);
    integer element3 = Int(Elem[result, e, esize], unsigned);
    integer res = element3 + Abs(element1 - element2);
    Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 | Zm |
1 1 0 0 0 0 0 0 | Zn  | Zda
```

SVE2

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>&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 + 0, esize DIV 2]);
  integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
  integer element3 = SInt(Elem[result, e, esize]);
  integer res = element3 + Abs(element1 - element2);
  Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
SABALT

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.

![](image)

SVE2

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.
<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.
TZn> Is the 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>

<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]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer res = element3 + Abs(element1 - element2);
    Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09 rc2_1, sve v2019-09 rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

<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>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></td>
</tr>
<tr>
<td>size</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

**SVE**

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.

<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>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

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 that conforms to all of the
following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand
    register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX instruction.
  • A predicated MOVPRFX instruction using the same governing predicate register and source element size as this
    instruction.
SABDLB

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.

```
<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>
```

SVE2

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>&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 + 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 |

SVE2

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.
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  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  |

SVE2

SADALP <Zda>, <T>, <Pg>/M, <Zn>, <Tb>

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

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 ElemP[mask, e, esize] == '0' then
    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]);
    integer element3 = SInt(Elem[operand_acc, e, esize]);
    Elem[result, e, esize] = (element1 + element2 + element3)<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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicate MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

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

SVE2

SADDLB <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);
sell = 0;
sel2 = 0;
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 + sell, 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.

```
| 0 1 0 0 0 1 0 1 | size | Zm | 1 0 0 0 0 0 0 | Zn | Zd |
```

**SVE2**

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.
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 0 1 0 1 | 0 | Zm | 0 0 0 0 1 | Zn | Zd |
```

**SVE2**

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.

| 31 30 29 28 27 26 25 24 23 22 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 0 0 0 0 0 0 1 | Pg | Zn | Vd |

SVE

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”:

<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>

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.

```
31 30 29 28 27 26 25 24 23 22 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 0 0 Zn Zd
```

SVE2

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;
bis(VL) operand1 = Z[n];
bis(VL) operand2 = Z[m];
bis(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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  |
```

SVE2

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);
```

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.
SBCLB

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | sz | 0  | Zm | 1  | 1  | 0  | 1  | 0  | 0  | Zn | Zda |

SVE2

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  | 1  | Zn | Zda |

SVE2

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.
<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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

```
0 1 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1
P <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

```
0 1 1 0 0 1 0 1 0 1 0 1 0 0 1 0 1
P <Zd>.H, <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 = 32;
integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR);

32-bit to single-precision

```
0 1 1 0 0 1 0 1 1 0 1 0 1 0 0 1 0 1 0 1
P <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 double-precision

```
0 1 1 0 0 1 1 0 1 0 0 1 0 1 0 1
P <Zd>.H, <Pg>/M, <Zn>.D
```

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

SCVTF <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 = FPRoundingMode(FPCR);

32-bit to double-precision

32-bit to double-precision

64-bit to half-precision

64-bit to half-precision

64-bit to single-precision

64-bit to single-precision
64-bit to single-precision

SCVTF <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 = FPRoundingMode(FPCR);

64-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 0 1 1 1 0 1 0 1 1 0 1 0 1 Pp Zn Zd

64-bit to double-precision

SCVTF <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 = FALSE;
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(Pp) 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 size 0 1 0 1 0 0 0 0 0 0 0 0 Pg Zm Zdn |

SDIV

SDIV <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 = 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<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 = 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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

SDIV
- 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 30 29 28 27 26 25 24 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 size 0 1 0 1 1 0 0 0 Pg Zm Zdn</td>
</tr>
</tbody>
</table>

**SVE**

SDIVR `<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 = 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<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 = 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**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 0 1 0 0 | size | 0 | Zm 0 0 0 0 0 0 | Zn | Zda

SVE

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

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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

```assembly
```

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

```assembly
SDOT <Zda>.D, <Zn>.H, <Zm>.H[i1]
```

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

```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 = SInt(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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

The instruction:

```
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</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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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).

Alien 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.
Initialise the first-fault register (FFR) to all true prior to a sequence of first-fault or non-fault loads. This instruction is unpredicated.

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

**SVE**

```java
if !HaveSVE() then UNDEFINED;
```

**Operation**

```java
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>31 30 29 28 27 26 25 24 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

**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><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 = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
0 1 0 0 0 1 0 1 0 tszh 1 tszl imm3 0 0 0 1 0 0 Zn Zd
```

SVE2

**SHRNB** <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.

<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>0</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>

<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>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] = 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.

| 31 30 29 28 27 26 25 24 23 22 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 1 | Zn | Zd |

**SVE2**

SHRNT <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 0 0</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0 1 x</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>S</td>
<td></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 0 0</td>
<td>RESERVED</td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>0 1 x</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>1 0 0</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 = 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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|     | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | Pg | Zm | Zdn|

SVE2

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

```
0 1 0 0 0 1 0 0 | size | 0 1 0 1 1 0 1 0 | Pg | Zm | Zdn
```

**SVE2**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

31 30 29 28 27 26 25 24 23 22 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 tszh 0 tszl imm3 1 1 1 0 1 Zn Zd

SVE2

SLI <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 = 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>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 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];
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  |

SVE2

SM4E <Zdn>.S, <Zdn>.S, <Zm>.S

if !HaveSVE2SM4() 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;
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  | 1  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 0  | Zm | 1  | 1  | 1  | 1  | 0  | 0  | Zn | 1  | 0  | Zd |

SVE2

SM4EKEY <Zd>.S, <Zn>.S, <Zm>.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 ROR 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.

```
0 0 0 0 0 1 0 0 | size | 0 0 1 0 0 0 0 0 | Pg | Zm | Zdn
```

**SVE**

```
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[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

<table>
<thead>
<tr>
<th>SMAX</th>
<th>SVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zdn</td>
<td>&lt;T&gt;</td>
</tr>
<tr>
<td>imm8</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td>1 0 1 0 0 0 1 1 0</td>
</tr>
</tbody>
</table>

**SMAX**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX 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 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

SVE2

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 = SI(e)[operand1, e + 0, esize];
      element2 = SI(e)[operand1, e + 1, esize];
    else
      element1 = SI(e)[operand2, e - 1, esize];
      element2 = SI(e)[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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | Pg | Zn | Vd |

**SVE**

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.

| 31 30 29 28 27 26 25 24 23 22 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 1 0 0 0 0 Pg Zm Zdn |

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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</td>
</tr>
<tr>
<td>Zdn</td>
</tr>
</tbody>
</table>

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
SMINP

Signed minimum pairwise.

Compute the minimum 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  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE2

SMINP <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.
<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.

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 = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

| 31 30 29 28 27 26 25 24 23 22 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 1 0 0 0 1 0 1 0 0 0 1 | Pg | Zn | Vd |

SVE

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.

<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

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();
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]);
  integer element3 = SInt(Elem[result, e, esize]);
  integer res = element3 + element1 * element2;
  Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.

An unpredicated MOVPRFX 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.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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**

| 31 30 29 28 27 26 25 24 23 22 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 | i3h | Zm | 1 0 0 0 | i3l | 0 | Zn | Zda |

**32-bit**


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 1 | i2h | Zm | 1 0 0 0 0 | i2l | 0 | Zn | Zda |

**64-bit**

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

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 res = element3 + element1 * element2;
    Elem[result, e, 2*esize] = res<2*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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX 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   | 1   | 0   | 0   | 1   | 0   | 0   | 0   | Zm  | 0   | 1   | 0   | 0   | 0   | 1   | Zn  | 0   | 1   | 0   | 0   | 0   | 1   |

SVE2

SMLALT <Zda>,<T>, <Zn>,<Tb>, <Zm>,<Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
i nteger esize = 8 << UInt(size);
i nteger n = UInt(Zn);
i nteger m = UInt(Zm);
i nteger 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();
i nteger 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]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer res = element3 + element1 * element2;
    Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | i3h | Zm  | 1  | 0  | 0  | 0  | i3l | Zn  | Zda |

**32-bit**

\[
\text{SMLALT } <\text{Zda}>.\text{S}, <\text{Zn}>.\text{H}, <\text{Zm}>.\text{H}[<\text{imm}>] \\
\]

If !\(\text{HaveSVE2}()\) then UNDEFINED;
integer esize = 16;
integer index = \(\text{UInt}(\text{i3h:i3l})\); integer n = \(\text{UInt}(\text{Zn})\);
integer m = \(\text{UInt}(\text{Zm})\);
integer da = \(\text{UInt}(\text{Zda})\);
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  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | i2h | Zm  | 1  | 0  | 0  | 0  | i2l | 1  | Zn  | Zda |

**64-bit**

\[
\text{SMLALT } <\text{Zda}>.\text{D}, <\text{Zn}>.\text{S}, <\text{Zm}>.\text{S}[<\text{imm}>] \\
\]

If !\(\text{HaveSVE2}()\) then UNDEFINED;
integer esize = 32;
integer index = \(\text{UInt}(\text{i2h:i2l})\); integer n = \(\text{UInt}(\text{Zn})\);
integer m = \(\text{UInt}(\text{Zm})\);
integer da = \(\text{UInt}(\text{Zda})\);
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);
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 res = element3 + element1 * element2;
    Elem[result, e, 2*esize] = res<2*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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  |

SVE2

SMLSLB <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 + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer res = element3 - element1 * element2;
    Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | i3h | Zm | 1  | 0  | 1  | i3l | Zn | Zda|
```

#### 32-bit


```c
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  | 1  | 1  | Zm | 1  | 0  | 1  | 0  | i2l | 0  | Zn | Zda|
```

#### 64-bit

SMLSLB \(<Zda>.D, <Zn>.S, <Zm>.S[<imm>]\)

```c
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]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer res = element3 - element1 * element2;
    Elem[result, e, 2*esize] = res<2*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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX instruction.
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.

```
0 1 0 0 0 1 0 0 | size | 0  Zm | 0 1 0 1 | 0 1 | Zn | Zda
```

**SVE2**

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]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer res = element3 - element1 * element2;
    Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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

<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>1</td>
<td>0</td>
<td>1</td>
<td>i3h</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i3l</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>
</tr>
</tbody>
</table>

32-bit


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>1</td>
<td>1</td>
<td>1</td>
<td>i2h</td>
<td>Zm</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>i2l</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>
</tr>
</tbody>
</table>

64-bit

SMLSLT <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]);
  integer element3 = SInt(Elem[result, e, 2*esize]);
  integer res = element3 - element1 * element2;
  Elem[result, e, 2*esize] = res<2*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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicate MOVPRFX 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.

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

Zm 0 1 0 0 1 0 1 0 0 0 1 0 1
Zn 1 0 0 1 1 0
Zda

SVE


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

If PSTATE.DIT is 1:

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

• The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
• 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 0 1 0 0 size | 1 | Zm | 0 1 1 0 1 0 | Zn | Zd |

SVE2

SMULH <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 = 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);
    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:
    o The values of the data supplied in any of its registers.
    o 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.
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.

```
<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</td>
</tr>
</tbody>
</table>
```

SVE2

SMULLB <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 + 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 response of this instruction to 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 (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

32-bit


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

64-bit

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

\begin{verbatim}
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;
\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.
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.

![Symbol Table]

**SVE2**

SMULLT \(<Zd>\), \(<T>\), \(<Zn>\), \(<Tb>\), \(<Zm>\)

if \(!\text{HaveSVE2}()\) then UNDEFINED;
if size == '00' then UNDEFINED;
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.

\(<Tb>\) 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}();
in integer elements = \text{VL} DIV esize;
in bits(\text{VL}) operand1 = Z[n];
in bits(\text{VL}) operand2 = Z[m];
in bits(\text{VL}) result;

for e = 0 to elements-1
  integer element1 = \text{SInt(Elem[operand1, 2*e + 1, esize DIV 2])};
in integer element2 = \text{SInt(Elem[operand2, 2*e + 1, esize DIV 2])};
in integer res = element1 * element2;
in \text{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.
**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

| 31 30 29 28 27 26 25 24 23 22 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 | i3h | Zm | 1 | 1 | 0 | 0 | i3l | 1 | Zn | Zd |

#### 32-bit

```
```

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 | 1 | 1 | 1 | i2h | Zm | 1 | 1 | 0 | 0 | i2l | 1 | Zn | Zd |

#### 64-bit

```
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

```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 = 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 |

Constructive

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 1 0 0 | Pg | Zm | Zdn |

Destructive

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”:

<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.
<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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
**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 | size | 0 0 1 0 0 0 1 0 1 | Pg | Zn | Zd
```

**SVE2**

SQABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```sql
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>&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**

```sql
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**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  |

**SVE2**

\[
\text{SQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>}
\]

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**

\[
\text{CheckSVEEnabled();}
\]

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 = 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**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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".

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24</th>
<th>size</th>
<th>1 0 0 1 0 0 1</th>
<th>sh</th>
<th>imm8</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

SVE

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

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

- The values of the NZCV flags.

- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

```
 0 0 0 0 0 1 0 0 size 1 | Zm 0 0 0 1 0 0 | Zn | Zd
```

SVE

SQADD <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;
```

Operational information

If PSTATE.DIT is 1:

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

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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 1  | 1  |  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

**SVE2**

SQCADD <Zdn>.<T>,<Zdn>.<T>,<Zm>.<T>,<const>

\[ \text{if } !\text{HaveSVE2}() \text{ then UNDEFINED}; \]
\[ \text{integer esize} = 8 \ll \text{UInt}(\text{size}); \]
\[ \text{integer m} = \text{UInt}(\text{Zm}); \]
\[ \text{integer dn} = \text{UInt}(\text{Zdn}); \]
\[ \text{boolean sub_i} = (\text{rot} == \text{'0'}); \]
\[ \text{boolean sub_r} = (\text{rot} == \text{'1'}); \]

**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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | pattern | Rdn |

#### 32-bit

```
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | pattern | Rdn |

#### 64-bit

```
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**

Page 2234
<pattern> | <pattern> |
<table>
<thead>
<tr>
<th></th>
<th></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 #uimm5</td>
<td></td>
</tr>
<tr>
<td>101x #uimm5</td>
<td></td>
</tr>
<tr>
<td>10110 #uimm5</td>
<td></td>
</tr>
<tr>
<td>1x010 #uimm5</td>
<td></td>
</tr>
<tr>
<td>1xx00 #uimm5</td>
<td></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);
```

**Operational information**

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

<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>0</td>
<td>imm4</td>
<td>1</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>
</tr>
</tbody>
</table>

#### 32-bit

```c
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

<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>imm4</td>
<td>1</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>
</tr>
</tbody>
</table>

#### 64-bit

```c
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":

---

**SQDECD (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>VL128</td>
</tr>
<tr>
<td>01011</td>
<td>VL256</td>
</tr>
<tr>
<td>01100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>01101</td>
<td>#uimm5</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>10000</td>
<td>#uimm5</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);
b_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);
```

**Operational information**

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

<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 1 1 1 0 imm4 1 1 0 0 1 0 pattern Zdn</td>
</tr>
</tbody>
</table>

**SVE**

```plaintext
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>000000</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>01101</td>
<td>VL256</td>
</tr>
<tr>
<td>01110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11001</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11011</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);
banner(VL) operand1 = Z[dn];
banner(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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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

| 31 30 29 28 27 26 25 24 23 22 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 0 | pattern | Rdn |

**32-bit**

```
SQDECH <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

| 31 30 29 28 27 26 25 24 23 22 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 1 | imm4 | 1 1 1 1 1 0 | pattern | Rdn |

**64-bit**

```
SQDECH <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":

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>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);
```

**Operational information**

If PSTATE.DIT is 1:

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

| 31 30 29 28 27 26 25 24 23 22 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 0 0 1 0 | pattern | Zdn |

**SVE**

```
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>01011</td>
<td>VL32</td>
</tr>
<tr>
<td>01101</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>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11100</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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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

![32-bit encoding](image)

```plaintext
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

![64-bit encoding](image)

```plaintext
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.
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);
```

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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  | 0  | 0  | 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  |
```

**SVE**

```
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”:
  - size `<T>`
    - 00 RESERVED
    - 01 H
    - 10 S
    - 11 D
- `<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**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  | 1  | 0  | imm4 | 1  | 1  | 1  | 1  | 1  | 0  | pattern | Rdn |

32-bit

```assembly
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  | 1  | 1  | imm4 | 1  | 1  | 1  | 1  | 1  | 0  | pattern | Rdn |

64-bit

```assembly
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":

SQDECW (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>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);
```

**Operational information**

If PSTATE.DIT is 1:

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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | Zdn |
```

SVE

```
SQDECW <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>000000</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>01110</td>
<td>VL128</td>
</tr>
<tr>
<td>01111</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>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 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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & Zm & 0 & 1 & 1 & 0 & 0 & 0 & 0 & Zn & 0 & 0 & 0 & 0 & 0 & Zda
\end{array}
\]

SVE

SQDMLALB <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);
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

\[
\text{CheckSVEEnabled}();
\]
\[
\text{integer elements} = \text{VL \ DIV \ esize};
\]
\[
\text{bits(VL)} \ \text{operand1} = Z[n];
\]
\[
\text{bits(VL)} \ \text{operand2} = Z[m];
\]
\[
\text{bits(VL)} \ \text{result} = Z[da];
\]
\[
\text{for } e = 0 \ \text{to elements-1}
\]
\[
\begin{align*}
\text{integer element1} &= \text{SInt}(\text{Elem}[\text{operand1, 2 * e + sel1, esize \ DIV \ 2}]); \\
\text{integer element2} &= \text{SInt}(\text{Elem}[\text{operand2, 2 * e + sel2, esize \ DIV \ 2}]); \\
\text{integer element3} &= \text{SInt}(\text{Elem}[\text{result, e, esize}]); \\
\text{integer product} &= \text{SInt(SignedSat}(2 * \text{element1} * \text{element2, esize}); \\
\text{Elem}[\text{result, e, esize}] &= \text{SignedSat}(\text{element3 + product, esize});
\end{align*}
\]
\[
Z[da] = \text{result};
\]
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.

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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

```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 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 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   |


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

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

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Zda&gt;</td>
<td>Is the name of the third source and destination scalable vector register, encoded in the “Zda” field.</td>
</tr>
<tr>
<td>&lt;Zn&gt;</td>
<td>Is the name of the first source scalable vector register, encoded in the “Zn” field.</td>
</tr>
<tr>
<td>&lt;Zm&gt;</td>
<td>For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the &quot;Zm&quot; field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the &quot;Zm&quot; field.</td>
</tr>
<tr>
<td>&lt;imm&gt;</td>
<td>For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the &quot;i3h:i3l&quot; fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the &quot;i2h:i2l&quot; fields.</td>
</tr>
</tbody>
</table>
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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
SQDMLALBT

Signed saturating doubling multiply-add long to accumulator (bottom \times 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.

| 31 30 29 28 27 26 25 24 23 22 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 0 0 0 1 0 | Zn | Zda |

SVE2

SQDMLALBT <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 = 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>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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

31 30 29 28 27 26 25 24 23 22 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 | Zm | 0 | 1 | 1 | 0 | 0 | 1 | Zn | Zda |

SVE2

SQDMLALT <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 = 1;
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”:

<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

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX instruction.
SQDMLALT (indexed)

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
```

32-bit

```
```

```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 1 i2h Zm 0 0 1 0 i2l 1 Zn Zda
```

64-bit

```
```

```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);

\( Z[da] = \text{result}; \)

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.

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Zm | 0  | 1  | 1  | 0  | 1  | 0  | Zn |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

SVE2

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></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></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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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 | 1 | i3l | 0 | Zn | Zda |

### 32-bit

```
```

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 | 1 | i2l | 0 | Zn | Zda |

### 64-bit

```
SQDMLSLB <Zda>.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 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} \text{ DIV}\ (2 * \text{esize});
integer eltspersegment = 128 \text{ DIV}\ (2 * \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 \text{ MOD}\ e\text{ltsperegment};
  integer element1 = \texttt{SInt(Elem[operand1, 2 * e + sel, esize])};
  integer element2 = \texttt{SInt(Elem[operand2, 2 * s + index, esize])};
  integer element3 = \texttt{SInt(Elem[result, e, 2*esize])};

  integer product = \texttt{SInt(SignedSat(2 * element1 * element2, 2*esize))};
  integer res = element3 - product;
  \texttt{Elem[result, e, 2*esize]} = \texttt{SignedSat(res, 2*esize)};
\end{verbatim}

\texttt{Z}[da] = \texttt{result};

Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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.

<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

SQDMLSLBT <Zda>.<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 da = \text{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”:

<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} \div \text{esize};
\text{bits(VL)}\text{ operand1} = Z[n];
\text{bits(VL)}\text{ operand2} = Z[m];
\text{bits(VL)}\text{ result} = Z[da];

for e = 0 to elements-1
  integer element1 = \text{SInt}([\text{Elem}\text{ operand1}, 2 * e + sel1, \text{esize} \text{ DIV } 2]);
  integer element2 = \text{SInt}([\text{Elem}\text{ operand2}, 2 * e + sel2, \text{esize} \text{ DIV } 2]);
  integer element3 = \text{SInt}([\text{Elem}\text{ result}, e, \text{esize}]);
  integer product = \text{SInt}([\text{SignedSat}\text{ (2 * element1 * element2, \text{esize})}]);
  \text{Elem}\text{ result, e, \text{esize} = SignedSat(element3 - product, \text{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 registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to 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 M0VPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The M0VPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The M0VPRFX instructions that can be used with this instruction are as follows:
• An unpredicated M0VPRFX 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  | 1  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
```

### SVE2

**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 sel1 = 1;
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":

```
<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**

If PSTATE.DIT is 1:

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

- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
**SQDMLSLT (indexed)**

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 0 1 0 0 1 0 1 i3h Zm 0 0 1 1 i3l Zn Zda</td>
</tr>
</tbody>
</table>

#### 32-bit

```assembly
```

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 0 1 0 0 1 1 1 i2h Zm 0 0 1 1 i2l Zn Zda</td>
</tr>
</tbody>
</table>

#### 64-bit

```assembly
```

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 = \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{SInt}(\texttt{Elem}[operand1, 2 * e + sel, esize]);
  integer element2 = \texttt{SInt}(\texttt{Elem}[operand2, 2 * s + index, esize]);
  integer element3 = \texttt{SInt}(\texttt{Elem}[result, e, 2*esize]);
  integer product = \texttt{SInt}(\texttt{SignedSat}(2 * element1 * element2, 2*esize));
  integer res = element3 - product;
  \texttt{Elem}[result, e, 2*esize] = \texttt{SignedSat}(res, 2*esize);
\texttt{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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Zm | 0  | 1  | 1  | 0  | 0  | Zn |  |  | Zd |  |

SVE2

SQDMULH $<Zd>.<T>,<Zn>.<T>,<Zm>.<T>$

if !HaveSVE2() then UNDEFINED;
integer esize = 8 $\ll$ 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 = $SInt(Elem[operand1, e, esize])$;
    integer element2 = $SInt(Elem[operand2, e, esize])$;
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res $\gg$ esize, 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.
**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

![16-bit encoding](image)


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

![32-bit encoding](image)

**SQDMULH <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

![64-bit encoding](image)

**SQDMULH <Zd>.D, <Zn>.D, <Zm>.D[^imm]**

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;

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;

Operational information

If PSTATE.DIT is 1:

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

• The response of this instruction to 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

```
0 1 0 0 0 1 0 1 | size 0 | Zm 0 1 1 0 0 0 | Zn | Zd
```

SVE2

```
SQDMULLB <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

```
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 = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res, 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.
**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>Zd</th>
<th>Zm</th>
<th>Zn</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000101i3h</td>
<td>110000 Zm</td>
<td>00000000 Zn</td>
</tr>
</tbody>
</table>

**32-bit**


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

<table>
<thead>
<tr>
<th>Zd</th>
<th>Zm</th>
<th>Zn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100010112h</td>
<td>110000 Zm</td>
<td>00000000 Zn</td>
</tr>
</tbody>
</table>

**64-bit**

**SQDMULLB** `<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

CheckSVEEnabled();
integer elements = VL \text{ DIV} (2 * esize);
integer eltspersegment = 128 \text{ 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 \text{ MOD} eltspersegment;
    integer element1 = \text{SInt}(\text{Elem}[operand1, 2 * e + sel, esize]);
    integer element2 = \text{SInt}(\text{Elem}[operand2, 2 * s + index, esize]);
    integer res = 2 * element1 * element2;
    \text{Elem}[result, e, 2*esize] = \text{SignedSat}(res, 2*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.
**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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | 0  | 1  | Zn |   |    |   |   |   |   |   |   |   |

**SVE2**

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**

```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
    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;
```

**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.
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

```
0 1 0 0 0 1 0 0 1 1 0 1 i3h  Zm 1 1 1 0 i3l 1  Zn  Zd
```

32-bit

```c
```

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

```
0 1 0 0 0 1 0 0 1 1 1 0 i2h  Zm 1 1 1 0 i2l 1  Zn  Zd
```

64-bit

```c
SQDMULLT <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 = 2 * element1 * element2;
    Elem[result, e, 2*esize] = SignedSat(res, 2*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.
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 | imm4 | 1 1 1 1 0 | 0 | pattern | Rdn
```

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 | 0 | pattern | Rdn
```

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>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);
```

**Operational information**

If PSTATE.DIT is 1:

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 0  | 0  | pattern| Rdn |

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | imm4| 1  | 1  | 1  | 1  | 0  | 0  | pattern| Rdn |

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":

SQINCD (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>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1x000</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>

<iimm> 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);
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 | imm4 | 1 | 1 | 0 | 0 | 0 | pattern | Zdn
```

**SVE**

**SQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}**

```c
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>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>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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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

31 30 29 28 27 26 25 24 23 22 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 | 0 | pattern | Rdn |

#### 32-bit

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

31 30 29 28 27 26 25 24 23 22 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 | 1 | imm4 | 1 | 1 | 1 | 1 | 0 | 0 | pattern | Rdn |

#### 64-bit

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)  
Page 2290
<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>101x1</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);
```

Operational information

If PSTATE.DIT is 1:

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

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.

```
31 30 29 28 27 26 25 24 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>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>
imm4 | 1 | 1 | 0 | 0 | 0 | 0 | pattern | Zdn
```

SVE

SQINCH \(<Zdn>\).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>01100</td>
<td>VL32</td>
</tr>
<tr>
<td>01101</td>
<td>VL64</td>
</tr>
<tr>
<td>01110</td>
<td>VL128</td>
</tr>
<tr>
<td>01111</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>1x01x</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
void 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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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

```
0 0 1 0 0 1 0 1 | size | 1 0 1 0 | 0 0 1 0 0 1 0 0 | Pm | Rdn
```

32-bit

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

```
0 0 1 0 0 1 0 1 | size | 1 0 1 0 | 0 0 1 0 0 1 0 0 | Pm | Rdn
```

64-bit

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);

Operational information

If PSTATE.DIT is 1:

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

| 31 30 29 28 27 26 25 24 23 22 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 | Pm | Zdn |

SVE

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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|imm4| 1  | 1  | 1  | 1  | 0  | 0  |pattern|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
Rdn
```

32-bit

```
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  | 1  | 0  | 0  | 1  | 0  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|imm4| 1  | 1  | 1  | 1  | 0  | 0  |pattern|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
Rdn
```

64-bit

```
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 = 32;
```

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":

SQINCW (scalar)
### Pattern

<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>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);
```

### Operational information

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

```
0 0 0 0 0 1 0 0 1 0 1 0 imm4 1 1 0 0 0 0 pattern Zdn
```

SVE

```
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>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

\textbf{CheckSVEEnabled}();
integer elements = \texttt{VL} \texttt{DIV} esize;
integer count = \texttt{DecodePredCount}(\texttt{pat}, esize);
bits(\texttt{VL}) operand1 = \texttt{Z}[dn];
bits(\texttt{VL}) result;

for \( e = 0 \) to \( \text{elements-1} \)
\begin{align*}
\text{integer element1} &= \texttt{Int}([\texttt{Elem}[\text{operand1}, e, \text{esize}], \text{unsigned}]) \\
(\texttt{Elem}[\text{result}, e, \text{esize}], -) &= \texttt{SatQ}(\text{element1} + (\text{count} \times \text{imm}), \text{esize}, \text{unsigned});
\end{align*}
\texttt{Z}[dn] = \text{result};

Operational information

If \text{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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is \texttt{UNPREDICTABLE}:
\begin{itemize}
  \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}
The \texttt{MOVPRFX} instructions that can be used with this instruction are as follows:
\begin{itemize}
  \item An unpredicated \texttt{MOVPRFX} instruction.
\end{itemize}
**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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 1  |
| size| Pg | Zn | Zd |
```

**SVE2**

```
SQNEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

if !HaveSVE2() then UNDEFINED;
integer esize = 8 \(<=\) \(\text{UInt}(\text{size})\);
integer g = \(\text{UInt}(\text{Pg})\);
integer n = \(\text{UInt}(\text{Zn})\);
integer d = \(\text{UInt}(\text{Zd})\);

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 = \(\text{VL} \div \text{esize}\);
bits(PL) mask = P[g];
bits(VL) operand = Z[n];
bits(VL) result = Z[d];
for e = 0 to elements-1
    integer element = \(\text{SInt}(\text{Elem}[\text{operand}, e, \text{esize}])\);
    if ElemP[mask, e, esize] \(\neq\) '1' then
        element = -element;
    Elem[result, e, esize] = \(\text{SignedSat}(\text{element}, \text{esize})\);
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The `MOVPRFX` instructions that can be used with this instruction are as follows:

- An unpredicated `MOVPRFX` instruction.
- A predicated `MOVPRFX` instruction using the same governing predicate register and source element size as 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.

```
31 30 29 28 27 26 25 24 23 22 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 0 1 1 | rot | Zn | Zda
```

SVE2

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

```cpp
CheckSVEEnabled();
integer pairs = VL \text{ 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);

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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  | 1  | 0  | 0  | 1  | i2 | Zm | 0  | 1  | 1  | 1  | rot | Zn | 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  | 1  | 0  | 0  | 1  | 1  | 1  | i1 | Zm | 0  | 1  | 1  | 1  | rot | Zn | Zda |

**16-bit**


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**

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>&lt;rot&gt;</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;

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

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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.

| 31 30 29 28 27 26 25 24 23 22 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 0 | Zm | 0 1 1 0 0 | Zn | Zda |

SVE2

SQRDMLAH \(<Zda>\).<T>, \(<Zn>\).<T>, \(<Zm>\).<T>

if \(\text{HaveSVE2}()\) then UNDEFINED;

integer isize = 8 \(<\text{UInt}(\text{size});
integer n = \text{UInt}(Zn);
integer m = \text{UInt}(Zm);
integer da = \text{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

\(\text{CheckSVEEnabled}();\)

integer elements = \(\text{VL} \div \text{esize};\)

\(\text{bits(VL)}\) operand1 = \(Z[n];\)
\(\text{bits(VL)}\) operand2 = \(Z[m];\)
\(\text{bits(VL)}\) operand3 = \(Z[da];\)

\(\text{bits(VL)}\) result;

integer round_const = 1 \(<\text{esize} - 1);\)

for \(e = 0 \) to elements-1

integer element1 = \(\text{SInt(Elem[operand1, e, esize]);}\)
integer element2 = \(\text{SInt(Elem[operand2, e, esize]);}\)
integer element3 = \(\text{SInt(Elem[operand3, e, esize]);}\)

integer res = (element3 \(<\text{esize}) + (2 \ast element1 \ast element2);

\(\text{Elem[result, e, esize]} = \text{SignedSat}((\text{res + round_const}) \gg \text{esize}, \text{esize});\)

\(Z[da] = \text{result};\)

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.

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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**

```
0 1 0 0 0 1 0 0 0 |i3h| i3l| Zm | 0 0 0 1 0 0 | Zn | Zda
```

**32-bit**

```
0 1 0 0 0 1 0 0 1 0 |i2| Zm | 0 0 0 1 0 0 | Zn | Zda
```

**64-bit**

```
0 1 0 0 0 1 0 0 1 1 |i1| Zm | 0 0 0 1 0 0 | Zn | 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

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
   • The MOVPRFX instruction must specify the same destination register as this instruction.
   • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
   • An unpredicated MOVPRFX instruction.
SQRDMMLSH (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 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | size | 0 | Zm | 0 | 1 | 1 | 1 | 0 | 1 | Zn | 0 | Zda |

SVE2

SQRDMMLSH <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);
   Elem[result, e, esize] = SignedSat((res + round_const) >> esize, 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 registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  | 0  | 0  | i3h | 1  | i3l | Zm | 0  | 0  | 0  | 1  | 0  | 1  | Zn |   |   |   |   |   |   |

16-bit

SQRDMLSH \(<\text{Zda}.\text{H}, \text{<Zn}.\text{H}, \text{<Zm}.\text{H}\[\text{<imm}>]>

if \(!\text{HaveSVE2}()\) 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  | 0  | i2 | Zm | 0  | 0  | 0  | 1  | 0  | 1  | Zn |   |   |   |   |   |   |

32-bit

SQRDMLSH \(<\text{Zda}.\text{S}, \text{<Zn}.\text{S}, \text{<Zm}.\text{S}\[\text{<imm}>]>

if \(!\text{HaveSVE2}()\) 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  | 1  | 1  | 1  | Zm | 0  | 0  | 0  | 1  | 0  | 1  | Zn |   |   |   |   |   |   |

64-bit

SQRDMLSH \(<\text{Zda}.\text{D}, \text{<Zn}.\text{D}, \text{<Zm}.\text{D}\[\text{<imm}>]>

if \(!\text{HaveSVE2}()\) 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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

### SVE2

SQRDMULH \(<Zd>.<T>, <Zn>.<T>, <Zm>.<T>\)

if !HaveSVE2() then UNDEFINED;
integer esize = 8 << \texttt{UInt} (size);
integer n = \texttt{UInt}(Zn);
ninteger m = \texttt{UInt}(Zm);
ninteger d = \texttt{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

\texttt{CheckSVEEnabled();}
integer elements = \texttt{VL} \div esize;
bits(\texttt{VL}) operand1 = \texttt{Z}[n];
bits(\texttt{VL}) operand2 = \texttt{Z}[m];
bits(\texttt{VL}) result;
ninteger round_const = 1 << (esize - 1);

for e = 0 to elements-1
    integer element1 = \texttt{SInt}([\texttt{Elem}[operand1, e, esize]]);
    integer element2 = \texttt{SInt}([\texttt{Elem}[operand2, e, esize]]);
    integer res = 2 * element1 \* element2;
    \texttt{Elem}[result, e, esize] = \texttt{SignedSat}((res + round_const) >> esize, esize);
\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 registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | i3h | i3l | Zm | 1  | 1  | 1  | 1  | 0  | 1  | Zn |   |   |   |   |   |

16-bit

\[
\text{SQRDMULH } <\text{Zd}.H, <\text{Zn}.H, <\text{Zm}.H[^{\text{imm}}] >}
\]

if !\texttt{HaveSVE2]() then UNDEFINED;
integer esize = 16;
integer index = \texttt{UInt}(\texttt{i3h:}\texttt{i3l});
integer n = \texttt{UInt}(\texttt{Zn});
integer m = \texttt{UInt}(\texttt{Zm});
integer d = \texttt{UInt}(\texttt{Zd});

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  | i2  | Zm | 1  | 1  | 1  | 1  | 0  | 1  | Zn |   |   |   |   |   |

32-bit

\[
\text{SQRDMULH } <\text{Zd}.S, <\text{Zn}.S, <\text{Zm}.S[^{\text{imm}}] >}
\]

if !\texttt{HaveSVE2()} then UNDEFINED;
integer esize = 32;
integer index = \texttt{UInt}(\texttt{i2});
integer n = \texttt{UInt}(\texttt{Zn});
integer m = \texttt{UInt}(\texttt{Zm});
integer d = \texttt{UInt}(\texttt{Zd});

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  | 1  | 1  | 1  | 0  | 1  | Zn |   |   |   |   |   |

64-bit

\[
\text{SQRDMULH } <\text{Zd}.D, <\text{Zn}.D, <\text{Zm}.D[^{\text{imm}}] >}
\]

if !\texttt{HaveSVE2()} then UNDEFINED;
integer esize = 64;
integer index = \texttt{UInt}(\texttt{i1});
integer n = \texttt{UInt}(\texttt{Zn});
integer m = \texttt{UInt}(\texttt{Zm});
integer d = \texttt{UInt}(\texttt{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;
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;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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  |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0   | 1  | 0  | 0  | 1  | 0  | 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  |

SVE2

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

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 ElemP[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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE2

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);

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[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] = SignedSat(res, esize);
  else
    Elem[result, e, esize] = Elem[operand2, 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
SQRSHRNBN

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 tszh 1 tszl imm3 0 0 1 0 1 0 Zn Zd

SVE2

SQRSHRNBN <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>00</td>
<td></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></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

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;
Operational information

If PSTATE.DIT is 1:

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

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.

<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 0</td>
</tr>
</tbody>
</table>

SVE2

SQRSHRNT <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>00</td>
<td>00</td>
<td>RESERVED</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>B</td>
</tr>
<tr>
<td>1x</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>xx</td>
<td>1x</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>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>xx</td>
<td>1x</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] = SignedSat(res, 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.
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.

<table>
<thead>
<tr>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1 0 tszh 1 tszl imm3 0 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

SVE2

SQRSHRUNB <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 0</td>
<td>0 0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1</td>
<td>B</td>
</tr>
<tr>
<td>0 1  x</td>
<td>1 x</td>
<td>H</td>
</tr>
<tr>
<td>1  x x</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 0</td>
<td>0 0</td>
<td>RESERVED</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0 1</td>
<td>H</td>
</tr>
<tr>
<td>0 1  x</td>
<td>1 x</td>
<td>S</td>
</tr>
<tr>
<td>1  x x</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] = UnsignedSat(res, esize);
  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.
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.

SVE2

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);

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 = (SInt(element) + round_const) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, 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.
**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.

<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 tszh 0 0 0 1 1 0 1 0 Pg tszl imm3 Zdn</td>
</tr>
</tbody>
</table>

**SVE2**

SQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```plaintext
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**

```plaintext
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 ElemP[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];
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

SVE2

```
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);

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);
  if ElemP[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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**SQSHLR**

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.

```
  31 30 29 28 27 26 25 24 23 22 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 1 1 0 0 1 0 0 | Pg | Zm | Zdn
```

**SVE2**

SQSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```c
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[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);
    if ElemP[mask, e, esize] == '1' then
        integer res = element << shift;
        Elem[result, e, esize] = SignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand2, 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 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   | 1   | 0   | 0   | Pg  | tszl | imm3 | Zdn |

**SVE2**

SQSHLU <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 = 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

**Asmble Symbol**

- `<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**

1. if !HaveSVE2() then UNDEFINED;
2. bits(3) tsze = tszh:tszl;
3. case tsze of
   when '000' UNDEFINED;
   when '001' esize = 8;
   when '01x' esize = 16;
   when '1xx' esize = 32;
4. integer n = UInt(Zn);
5. integer d = UInt(Zd);
6. integer shift = (2 * esize) - UInt(tsz:imm3);
7. integer elements = VL DIV (2 * esize);
8. bits(VL) operand = Z[n];
9. bits(VL) result;
10. for e = 0 to elements-1
    11.   bits(2*esize) element = Elem[operand, e, 2*esize];
    12.   integer res = SInt(element) >> shift;
    13.   Elem[result, 2*e + 0, esize] = SignedSat(res, esize);
    14.   Elem[result, 2*e + 1, esize] = Zeros();
15. 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.
**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.

| 31 30 29 28 27 26 25 24 23 22 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 | 1 | 0 | 0 | 1 | Zn | Zd |

**SVE2**

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(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>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 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 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 = 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;

SQSHRNT
Operational information

If PSTATE.DIT is 1:

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

---

**SVE2**

SQSHRUNB <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”:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tszh</td>
<td>tszl</td>
<td>&lt;T&gt;</td>
</tr>
<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”:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tszh</td>
<td>tszl</td>
<td>&lt;Tb&gt;</td>
</tr>
<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] = UnsignedSat(res, esize);
  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.
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.

<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></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></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>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>01</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>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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
**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 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 1 0 1 0 0 | Pg | Zm | Zdn |

**SVE2**

\[
\text{SQSUB} <\text{Zdn}>, <\text{T}>, <\text{Pg}>/M, <\text{Zdn}>, <\text{T}>, <\text{Zm}>, <\text{T}>
\]

if \(!\text{HaveSVE2}()\) 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 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;\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 elements = \text{VL} \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{SInt}(\text{Elem}[operand1, e, esize]);
integer element2 = \text{SInt}(\text{Elem}[operand2, e, esize]);
if \(\text{Elem}[mask, e, esize] == '1'\) then

integer res = \text{SInt}(\text{Sat}(element1 - element2, esize, unsigned));
\text{Elem}[result, e, esize] = res\langle\text{esize-1:0}\rangle;

else

\text{Elem}[result, e, esize] = \text{Elem}[operand1, e, esize];

\text{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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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".

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | sh | imm8 | Zdn |

**SVE**

SQSUB <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**

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

The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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.

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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   | 1   | 0   | 0   | 0   | 1   | 1   | 0   | 0   | 0   | 0   | 1   | 0   | 0   | 0   | 0   | 0   |
```

SVE

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;

Operational information

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

<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

SQSUBR \(<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(P[0]) 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 = 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand
    register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

  • An unpredicated MOVPRFX instruction.
  • A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

```
0 1 0 0 0 1 0 1 0 | tszh 1 0 0 0 0 1 0 0 0 | Zn 1 0 0 1 0 1 0 0 0
```

**SVE2**

SQXTNB `<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>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 = SignedSat(element1, halfesize);
    Elem[result, 2*e + 0, halfesize] = res;
    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.
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.

```
0 1 0 0 0 1 0 1 0 | tszh | 1 tszl | 0 0 0 0 1 0 0 0 | Zn | Zd
```

SVE2

SQXTNT \(<\text{Zd}>.\text{<T>}, \ <\text{Zn}>.\text{<Tb>}\)

if !\text{HaveSVE2}() then UNDEFINED;

bits(3) tsize = tszh:tszl;

```latex
\text{case tsize of }
\begin{align*}
\text{when '001' esize} & = 16; \\
\text{when '010' esize} & = 32; \\
\text{when '100' esize} & = 64; \\
\text{otherwise UNDEFINED;}
\end{align*}
```

integer n = \text{UInt}(\text{Zn});

integer d = \text{UInt}(\text{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 “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>10</td>
<td>H</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>S</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

\(<\text{Zn}>\) Is the name of the first source scalable vector register, encoded in the “Zn” field.

\(<\text{Tb}>\) 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>00</td>
<td>01</td>
<td>H</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>S</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>D</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>RESERVED</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Operation**

\text{CheckSVEEnabled}();

integer elements = \text{VL DIV esize};

\text{bits(VL) operand1 = Z[n];}

\text{bits(VL) result = Z[d];}

integer halfesize = esize DIV 2;

for e = 0 to elements-1

\begin{align*}
\text{integer element1} & = \text{SInt}(\text{Elem}[\text{operand1, e, esize}]); \\
\text{bits(halfesize) res} & = \text{SignedSat}(\text{element1, halfesize}); \\
\text{Elem}[\text{result, Z[e + 1, halfesize]}] & = \text{res};
\end{align*}

\text{Z[d] = result;}

SQXTNT
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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.

![Binary representation](image)

**SVE2**

SQXTUNB `<Zd>.<T>, <Zn>.<Tb>`

if `!HaveSVE2()` then UNDEFINED;

```plaintext
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”:
  ```plaintext
<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 source scalable vector register, encoded in the “Zn” field.
- `<Tb>` Is the size specifier, encoded in “tszh:tszl”:
  ```plaintext
<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;
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;

Operational information

If PSTATE.DIT is 1:

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

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.

![Binary representation of destination elements]

### SVE2

```plaintext
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”:
  ```plaintext
<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”:
  ```plaintext
<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 = UnsignedSat(element1, halfesize);
  Elem[result, 2*e + 1, halfesize] = res;

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.
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 size 0 1 0 1 0 0 1 0 0 Pg Zm Zdn

SVE2

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

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 that conforms to all of the
following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand
  register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 | 0 | 1 | 0 | 1 | tszh | 0 | tszl | imm3 | 1 | 1 | 1 | 1 | 0 | 0 | Zn | Zd |

SVE2

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>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($VL$) 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 response of this instruction to asynchronous exceptions does not vary based on:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
**SRSHL**

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  | 0  | 1  | 0  | 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  |

<table>
<thead>
<tr>
<th>size</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
</table>

**SVE2**

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 ElemP[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];

end if

end for

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 \texttt{MOVPRFX} instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is \texttt{UNPREDICTABLE}:
- 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.

The \texttt{MOVPRFX} instructions that can be used with this instruction are as follows:
  - An unpredicated \texttt{MOVPRFX} instruction.
  - A predicated \texttt{MOVPRFX} instruction using the same governing predicate register and source element size as this instruction.
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.

### 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 = 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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 |
| 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  |   |   |   |   |
| tzh | tszl | Pg | tszl | imm3 | Zdn |

**SVE2**

SRSHR <Zdn>.<T>, <Pg>/M, <Zdn>.<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 g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tszl: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 “tzh:tszl”:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| tzh | tszl |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 00  | 00  | RESERVED |
| 00  | 01  | B        |
| 00  | 1x  | H        |
| 01  | xx  | S        |
| 1x  | xx  | D        |

<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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 |
|-----------------------------------------------|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|-- |--}
• The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX 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 1 0 1 0 tszh 0 tszl imm3 1 0 1 0 0 0 Zn Zd |

SVE2

SSHLLB <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</td>
<td>0</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>

<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>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);
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 = 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.
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.

<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 0 tszh 0 tszl imm3 1 0 1 0 0 1</td>
</tr>
</tbody>
</table>

SVE2

SSHLLT $<$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</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>

$<$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>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);

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 = $<$Int (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.
SSRA

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.

\[
\begin{array}{cccccccccccccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & tzh & 0 & tszl & imm3 & 1 & 1 & 1 & 0 & 0 & 0 & Zn & Zda
\end{array}
\]

SVE2

SSRA <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 “tsz: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;
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
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 0 1 0 1 | size | 0 | Zm | 0 0 0 1 0 0 | Zn | Zd |

SVE2

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;

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.

<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.
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.

SVE2

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 0 1 0 1 | Zn | Zd |
```

SVE2

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.
SSUBLTB

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.

```
31 30 29 28 27 26 25 24 23 22 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 0 0 1 1 | Zn | Zd
```

SVE2

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.

<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, 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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  |

SVE2

SSUBWB <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.
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>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 1 size 0 Zm 0 1 0 1 0 1 Zn Zd</td>
</tr>
</tbody>
</table>
```

SVE2

SSUBWT <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”:
  - `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.
- `<Zm>` Is the name of the second source scalable vector register, encoded in the "Zm" field.
- `<Tb>` Is the size specifier, encoded in “size”:
  - `size`<Tb>
    - 00: RESERVED
    - 01: B
    - 10: H
    - 11: S

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

32-bit element

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

64-bit element

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

```c
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 SetNotTagCheckedInstruction(FALSE);
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>;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

```
| 31 30 29 28 27 26 25 24 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 | size | imm4 | Pg | Rn | Zt |
```

SVE

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 |
|--------------------------------------------------|---------------------------------|---------------------------------|---------------------------------|---------------------------------|
| 1 1 1 0 0 1 0 0 0 | size | Rm | 0 | 1 | 0 | Pg | Rn | Zt |

SVE

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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

```
31 30 29 28 27 26 25 24 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 Zm 1 xs 0 Pg Rn Zt
```

### 32-bit unpacked unscaled offset

```
ST1B { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]
```

```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 = 8;
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 0 1 0 Zm 1 xs 0 Pg Rn Zt
```

### 32-bit unscaled offset

```
ST1B { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]
```

```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 = 8;
integer offs_size = 32;
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 1 0 0 1 0 0 0 0 Zm 1 0 1 Pg Rn Zt
```

64-bit unscaled offset

```
ST1B (scalar plus vector)
64-bit unscaled offset

STIB { <Zt>.D }, <Pg>, [<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 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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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>;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09.rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 |

**SVE**

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 SetNotTagCheckedInstruction(FALSE);

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>;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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 | size | 0 | imm4 | 1 | 1 | 1 | Pg | Rn | Zt |

**SVE**

```
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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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.

SVE

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | Zm | 1  | xs | 0  | Pg | Rn | Zt |

32-bit unpacked unscaled offset

```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 = 64;
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  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 1  | Zm | 1  | 0  | 1  | Pg | Rn | Zt |

64-bit unscaled offset

```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 = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 8;
```
64-bit scaled offset

```
ST1D \{ <Zt>.D \}, <Pg>, [<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 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 0 1 1 1 0 0 | Zm | 1 0 1 | Pg | Rn | Zt
```

64-bit unscaled offset

```
ST1D \{ <Zt>.D \}, <Pg>, [<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 offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Zt&gt;</td>
<td>Is the name of the scalable vector register to be transferred, encoded in the “Zt” field.</td>
</tr>
<tr>
<td>&lt;Pg&gt;</td>
<td>Is the name of the governing scalable predicate register P0-P7, encoded in the &quot;Pg&quot; field.</td>
</tr>
<tr>
<td>&lt;Xn</td>
<td>SP&gt;</td>
</tr>
<tr>
<td>&lt;Zm&gt;</td>
<td>Is the name of the offset scalable vector register, encoded in the &quot;Zm&quot; field.</td>
</tr>
<tr>
<td>&lt;mod&gt;</td>
<td>Is the index extend and shift specifier, encoded in “xs”:</td>
</tr>
</tbody>
</table>

<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} \texttt{esize};
bits(64) base;
bits(\texttt{VL}) offset = \texttt{Z}[m];
bits(\texttt{VL}) src = \texttt{Z}[t];
bits(\texttt{PL}) mask = \texttt{P}[g];
bits(64) addr;
constant integer mbytes = \texttt{msize} \texttt{DIV} 8;

if \texttt{HaveMTEExt}() then SetNotTagCheckedInstruction(FALSE);

if \texttt{n} == 31 then
    \texttt{CheckSPAlignment}();
    base = \texttt{SP}[];
else
    base = \texttt{X}[n];

for \texttt{e} = 0 to elements-1
    if \texttt{ElemP[mask, e, esize]} == '1' then
        integer off = \texttt{Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned});
        addr = base + (off << scale);
        \texttt{Mem[addr, mbytes, AccType_NORMAL]} = \texttt{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

```
0 0 0 0 0 1 0 1 1 1 | imm5 | 1 0 1 | Pg | Zn | Zt
```

### 64-bit element

```
0 0 0 0 1 0 0 1 1 1 | imm5 | 1 0 1 | Pg | Zn | Zt
```

#### 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.

**ST1H (vector plus immediate)**

Page 2409
Operation

$\text{CheckSVEEnabled}();$
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 SetNotTagCheckedInstruction(FALSE);

for e = 0 to elements-1
    if $\text{ElemP[mask, e, esize]} == '1'$ then
        addr = $\text{ZeroExtend}(\text{Elem}[\text{base, e, esize}], 64) + \text{offset} \times \text{mbytes};$
        $\text{Mem[addr, mbytes, AccType_NORMAL]} = \text{Elem[src, e, esize]}_{<\text{msize}-1:0>};$

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.

| 31 30 29 28 27 26 25 24 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  size 0 | imm4  1  1  1                  | Pg  Rn  Zt |

SVE

ST1H { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 8 << UInt(size);
integer msize = 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 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.
<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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
**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  | 0  | 1  | size | Rm | 0  | 1  | 0  | Pg | Rn | Zt |

**SVE**

ST1H { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```plaintext
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 = X[m];
bits(VL) src = Z[t];
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
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

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 esize = 32;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked scaled offset

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 esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked unscaled offset

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 esize = 64;
integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;

32-bit unpacked unscaled offset
32-bit unpacked unscaled offset

```
STI H {<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 size = 64;
integer msize = 16;
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 0 | 1 1 0 | Zm 1 xs 0 Pg Rn Zt
```

32-bit unscaled offset

```
STI H {<Zt>.S}, <Pg>, [<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 size = 32;
integer msize = 16;
integer offs_size = 32;
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 1 0 0 1 0 0 | 0 1 0 1 | Zm 1 0 1 Pg Rn Zt
```

64-bit scaled offset

```
STI H {<Zt>.D}, <Pg>, [<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 size = 64;
integer msize = 16;
integer offs_size = 64;
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 1 0 0 1 0 | 0 1 0 0 | Zm 1 0 1 Pg Rn Zt
```

ST1H (scalar plus vector)
64-bit unscaled offset

STIH { <Zt>.D }, <Pg>, [<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 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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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>;

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  | 0  | 1  | 1  | imm5| 1  | 0  | 1  | Pg | Zn | Zt |

32-bit element

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  | 1  | 0  | imm5| 1  | 0  | 1  | Pg | Zn | Zt |

64-bit element

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 SetNotTagCheckedInstruction(FALSE);

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 | size | 0 | imm4 | 1 | 1 | 1 | Pg | Rn | Zt |

**SVE**

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
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 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | size | Rm | 0  | 1  | 0  | Pg | Rn | Zt |

SVE

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
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

![32-bit scaled offset diagram]

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

![32-bit unpacked scaled offset diagram]

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

![32-bit unpacked unscaled offset diagram]

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 unscaled offset

STIW {<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 = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

32-bit unscaled offset

32-bit unscaled offset

64-bit scaled 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 = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;

64-bit scaled 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 = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset

32-bit unscaled offset

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 = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;

64-bit unscaled offset
64-bit unscaled offset

ST1W { <Zt>.D }, <Pg>, [<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 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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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);
        Mem[addr, mbytes, AccType_NORMAL] = Elem[src, e, esize]<msize-1:0>;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09 rc2_1, sve v2019-09 rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 11 | 11 | 11 | imm4| 1  | 1  | 1  | Pg |   |   | Rn |   |   |   |   |   | Zt |

SVE

SVE

ST2B { <Zt1>.B, <Zt2>.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);
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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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

SVE

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

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
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.

SVE

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);
innteger 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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[ ];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 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.

### SVE

ST2D \{ <Zt1>.D, <Zt2>.D \}, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```assembly
if !HaveSVE() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
nInteger n = UInt(Rn);
mInteger m = UInt(Rm);
gInteger g = UInt(Pg);
integer esize = 64;
nInteger 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 HaveMTEEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
```
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.

<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>1</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>
<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>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE

ST2H { <Zt1>.H, <Zt2>.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);
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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[ ];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
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.

SVE

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

<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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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.

31 30 29 28 27 26 25 24 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 1 imm4 1 1 1 Pg Rn Zt

SVE

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

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
  CheckSAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[ ];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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 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   |   0   |   1   |   0   |   1   |   1   |   0   |   1   |   1   |   0   |   1   |   0   |   1   |   0   |   1   |   0   |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|   Rm  |   Pg  |   Rn  |   Zt  |

**SVE**

SVE

\[
\text{ST2W \{ <Zt1>.S, <Zt2>.S \}, <Pg>, [<Xn|SP>, <Xm>, LSL #2]}\]

if \(!\text{HaveSVE}()\) then UNDEFINED;
if \(\text{Rm} == '11111'\) then UNDEFINED;
integer \(t = \text{UInt}(\text{Zt})\);
integer \(n = \text{UInt}(\text{Rn})\);
integer \(m = \text{UInt}(\text{Rm})\);
integer \(g = \text{UInt}(\text{Pg})\);
integer \(\text{esize} = 32\);
integer \(\text{nreg} = 2\);

**Assembler Symbols**

\(<\text{Zt1}>\) Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
\(<\text{Zt2}>\) 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.
\(<\text{Xn|SP}>\) 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(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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
```
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.

```
ST3B (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 = 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
  CheckSPAlignment();
  if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
  base = SP[];
else
  if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
  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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

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 HaveMTEEExt() then SetNotTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
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;
```
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.

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & \text{imm4} & 1 & 1 & 1 & \text{Pg} & \text{Rn} & \text{Zt} \\
\end{array}
\]

SVE

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

```plaintext
CheckSVEEnabled();
integer elements = VL \text{ DIV} esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g];
constant integer mbytes = esize \text{ DIV} 8;
array [0..2] of bits(VL) values;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    base = X[n];

for r = 0 to nreg-1
    values[r] = 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 ElemP[mask, e, esize] == '1' then
            Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];
        addr = addr + mbytes;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

<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 1 0 1 1 1 0 1 0 1 1 1 0 0 1 1 1 0</td>
</tr>
</tbody>
</table>

SVE

ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.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 = 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

\texttt{CheckSVEEnabled();}
integer elements = \texttt{VL} \div \texttt{esize};
bites(64) base;
bites(64) addr;
bites(PL) mask = P[g];
bites(64) offset = X[m];
constant integer mbytes = esize \div 8;
array [0..2] of bites(\texttt{VL}) values;

if \texttt{HaveMTEExt()} then SetNotTagCheckedInstruction(FALSE);

if \texttt{n == 31} then
    \texttt{CheckSPLignment();}
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 + \texttt{UInt}(offset) * mbytes;
    for r = 0 to nreg-1
        if \texttt{ElemP[mask, e, esize]} == '1' then
            \texttt{Mem[addr, mbytes, AccType_NORMAL] = Elem[values[r], e, esize];}
        addr = addr + mbytes;
        offset = offset + nreg;
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.

<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 imm4 1 1 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

SVE

\[
\text{ST3H \{ <Zt1>.H, <Zt2>.H, <Zt3>.H \}, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]}
\]

if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
integer esize = 16;
integer offset = SInt(imm4);
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.
\(<\text{imm}>\) 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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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.

<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>Rm</th>
<th>0 1 1</th>
<th>Pg</th>
<th>Rn</th>
<th>Zt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0 1 0</td>
<td>1 1 0</td>
<td>0 1 1</td>
<td>1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SVE**

```assembly
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

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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;
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.

### 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.

```assembly
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

```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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    base = X[n];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32];

addr = base + offset * elements * nreg * mbytes;
for r = 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;
```
**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]
```

SVE

```
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 = \texttt{VL} \div \texttt{esize};
bits(64) \texttt{base};
bits(64) \texttt{addr};
bits(\texttt{PL}) \texttt{mask} = \texttt{P}[g];
bits(64) \texttt{offset} = \texttt{X}[m];
constant integer \texttt{mbytes} = \texttt{esize} \div 8;
array [0..2] of bits(\texttt{VL}) \texttt{values};

if \texttt{HaveMTEExt()} then SetNotTagCheckedInstruction(FALSE);

if \texttt{n == 31} then 
  CheckSPAlignment();
else
  \texttt{base} = \texttt{X}[n];

for \texttt{r = 0} to \texttt{nreg-1}
  \texttt{values}[r] = \texttt{Z}[(t+r) \mod 32];

for \texttt{e = 0} to \texttt{elements-1}
  \texttt{addr} = \texttt{base} + \texttt{UInt}(\texttt{offset}) \times \texttt{mbytes};
  for \texttt{r = 0} to \texttt{nreg-1}
    if \texttt{ElemP}[\texttt{mask}, \texttt{e}, \texttt{esize}] == '1' then
      \texttt{Mem}[\texttt{addr}, \texttt{mbytes}, \texttt{AccType\_NORMAL}] = \texttt{Elem}[\texttt{values}[r], \texttt{e}, \texttt{esize}];
    \texttt{addr} = \texttt{addr} + \texttt{mbytes};
  \texttt{offset} = \texttt{offset} + \texttt{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.

```
 31 30 29 28 27 26 25 24 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 1 1 imm4 1 1 1 Pg Rn Zt
```

SVE


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(PL) mask = P[g];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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
1 1 1 0 0 1 0 0 1 1  Rm 0 1 1  Pg  Rn  Zt

SVE


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 SetNotTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

<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 1 1 imm4 1 1 1 Pg Rn Zt</td>
</tr>
</tbody>
</table>

SVE


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

\begin{verbatim}
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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
\end{verbatim}
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 1         | Rm              | Pg              |
| 0 1 1                         | Rn              | Zt              |

SVE

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 `HaveMTEEExt()` then SetNotTagCheckedInstruction(FALSE);

if `n == 31` then
    `CheckSPAlignment();`
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;
**ST4H (scalar plus immediate)**

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.

```
```

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

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 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.

31 30 29 28 27 26 25 24 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 Rm 0 1 1 Pg Rn Zt

SVE


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 = 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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 | 1 | 1 | imm4 | 1 | 1 | 1 | Pg | Rn | Zt |

SVE


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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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;
```
ST4W (scalar plus scalar)

Contiguous store four-word structures from four vectors (scalar 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 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.

| 31 30 29 28 27 26 25 24 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  1 | Rm | 0  1  1 | Pg | Rn | Zt |

SVE

ST4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.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 = 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 = \textit{VL} \text{ DIV} \text{ esize};

bits(64) base;
bits(64) addr;

bits(PL) mask = \textit{P}[g];
bits(64) offset = \textit{X}[m];
constant integer mbytes = esize \text{ DIV} 8;
array [0..3] of bits(VL) values;

if \textit{HaveMTEExt}() then SetNotTagCheckedInstruction(FALSE);

if n == 31 then
  \textbf{CheckSPAlignment}();
  base = \textit{SP}[];
else
  base = \textit{X}[n];

for r = 0 to \text{nreg-1}
  values[r] = \textit{Z}[(t+r) \text{ MOD} 32];

for e = 0 to \text{elements-1}
  addr = base + UInt(\text{offset}) \times \text{mbytes};
  for r = 0 to \text{nreg-1}
    if \text{ElemP}[mask, e, esize] == '1' then
      \text{Mem}[addr, \text{mbytes}, \text{AccType_NORMAL}] = \text{Elem}[values[r], e, esize];
      addr = addr + mbytes;
  offset = offset + nreg;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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  | Rm | 0  | 0  | 1  | Pg | Zn | Zt |

### 32-bit unscaled offset

```
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  | Rm | 0  | 0  | 1  | Pg | Zn | Zt |

### 64-bit unscaled offset

```
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

\texttt{CheckSVEEnabled();}
\begin{verbatim}
integer elements = \texttt{VL} \texttt{DIV} \texttt{esize};
bits(\texttt{VL}) base = \texttt{Z[n]};
bits(64) offset = \texttt{X[m]};
bits(\texttt{VL}) src = \texttt{Z[t]};
bits(\texttt{PL}) mask = \texttt{P[g]};
bits(64) addr;
constant integer mbytes = \texttt{msize DIV 8};
if \texttt{HaveMTEExt()} then SetNotTagCheckedInstruction(\texttt{FALSE});
\end{verbatim}

\begin{verbatim}
for e = 0 to elements-1
  if \texttt{ElemP[mask, e, esize]} == '1' then
    addr = \texttt{ZeroExtend(Elem[base, e, esize], 64)} + offset;
    \texttt{Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize]<msize-1:0>};
\end{verbatim}

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

### SVE

```c
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

```c
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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 © 2010-2019 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.

1 1 1 1 0 0 1 0 | 0 | 0 | 0 | Rm | 0 | 1 | 1 | Pg | Rn | Zt

SVE

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

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 HaveMTEEExt() then SetNotTagCheckedInstruction(FALSE);
if n == 31 then
    CheckSPAlignment();
    base = SP[];
exthilese
    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;
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.

SVE2

STNT1D { <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 = 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.
<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 SetNotTagCheckedInstruction(FALSE);
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.

```
<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>0</td>
<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>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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

SVE

```
STNT1D { <Zt>.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);

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
STNT1D (scalar plus scalar)

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.

```
31 30 29 28 27 26 25 24 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 0 1 1 0 1 1 0 Rm 0 1 1 Pg Rn Zt
```

SVE

```
STNTD { <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;

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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
```

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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>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>1</td>
<td>1</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></td>
</tr>
</tbody>
</table>

#### 32-bit unscaled offset

```plaintext
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>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>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>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

#### 64-bit unscaled offset

```plaintext
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 = \texttt{VL} \text{ DIV} \text{ esize};
bits(\texttt{VL}) base = \texttt{Z}[n];
bits(64) offset = \texttt{X}[m];
bits(\texttt{VL}) src = \texttt{Z}[t];
bits(\texttt{PL}) mask = \texttt{P}[g];
bits(64) addr;
constant integer mbytes = \text{ msize} \text{ DIV} 8;

if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);

for e = 0 to elements - 1
  if ElemP[mask, e, esize] == '1' then
    addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
    Mem[addr, mbytes, \text{ AccType\_STREAM}] = Elem[src, e, esize]<\text{ 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

SVE

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
    CheckSPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    base = X[n];
src = Z[t];

addr = base + offset * elements * mbytes;
for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1'
        Mem[addr, mbytes, AccType_STREAM] = Elem[src, e, esize];
    addr = addr + mbytes;

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 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  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | Rm |
| 0  | 1  | 1  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Pg |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Rn |
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | Zt |

SVE

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 SetNotTagCheckedInstruction(FALSE);

if n == 31 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 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

![Binary representation of 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>1</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>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>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

![Binary representation of 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>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>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>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

\[
\begin{align*}
\text{CheckSVEEnabled}(); \\
\text{integer elements} &= \mathbf{VL} \div \mathbf{esize}; \\
\text{bits(}\mathbf{VL}\text{) base} &= \mathbf{Z}[n]; \\
\text{bits(64)} \text{ offset} &= \mathbf{X}[m]; \\
\text{bits(}\mathbf{VL}\text{) src} &= \mathbf{Z}[t]; \\
\text{bits(}\mathbf{PL}\text{) mask} &= \mathbf{P}[g]; \\
\text{bits(64)} \text{ addr} &= ; \\
\text{constant integer mbytes} &= \mathbf{msize} \div \mathbf{8}; \\
\text{if } \text{HaveMTEExt()} \text{ then SetNotTagCheckedInstruction(FALSE);} \\
\text{for } e = 0 \text{ to elements-1} \\
\quad \text{if } \text{ElemP[mask, e, esize]} = '1' \text{ then} \\
\qquad \text{addr} &= \text{ZeroExtend(Elem[base, e, esize], 64)} + \text{offset}; \\
\qquad \text{Mem[addr, mbytes, AccType_STREAM]} &= \text{Elem[src, e, esize]<msize-1:0>};
\end{align*}
\]
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.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.

---

SVE

STNT1W { <Zt>.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);

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
    CheckSPPAlignment();
    if HaveMTEExt() then SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 © 2010-2019 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.

<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>Rm   0 1 1         Pg       Rn   Zt</td>
</tr>
</tbody>
</table>

SVE

STNT1W { <Zt>, <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;
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 SetNotTagCheckedInstruction(FALSE);
if n == 31 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;
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>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>
</tbody>
</table>

imm9h   0  0  imm9l   Rn   0  Pt

SVE

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 SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | imm9h | 0  | 1  | 0  | imm9l | Rn |    |    |    |    |    |    |    |    |    |    |  Zt |
```

**SVE**

```
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 SetNotTagCheckedInstruction(TRUE);
    base = SP[];
else
    if HaveMTEExt() then SetNotTagCheckedInstruction(FALSE);
    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 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><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

```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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 "#<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
**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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Zm | 0  | 0  | 0  | 0  | 1  | Zn | Zd |

**SVE**

SVE `<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**

```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] = 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.
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.

<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</td>
</tr>
</tbody>
</table>

SVE2

SUBHNB <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 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 + 0, halfsize] = res<halfsize-1:0>;
    Elem[result, 2*e + 1, halfsize] = 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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0  | 1  | 0  | 0  | 1  | 0  | 1  | Zm | 0  | 1  | 1  | 0  | 1  | Zn | 0  | 1  | 1  | 0  | 1  | Zd |

SVE2

SUBHNT <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;
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  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  |

SVE

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;
bots(PL) mask = P[g];
bots(VL) operand1 = Z[dn];
bots(VL) operand2 = Z[m];
bots(VL) result;
for e = 0 to elements-1
    bots(esize) element1 = Elem[operand1, e, esize];
    bots(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 |
| 0 0 1 0 0 1 0 1 | size | 1 0 0 | 0 1 | 1 1 | 1 | sh | imm8 | Zdn |

SVE

SUBR <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;

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 = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
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.

SVE


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

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

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

- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 0 1 0 1 | size 1 1 0 0 0 0 1 1 1 0 | Zn | Zd
```

**SUNPKHI** `<Zd>.<T>, <Zn>.<Tb>`

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 0 1 0 1 | size 1 1 0 0 0 0 1 1 1 0 | Zn | Zd
```

**SUNPKLO** `<Zd>.<T>, <Zn>.<Tb>`

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

```plaintext
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.
SUQADD

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.

31 30 29 28 27 26 25 24 23 22 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 | 0 | 1 | 0 | 0 | Pg | Zm | Zdn |

SVE

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 ElemP[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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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**

<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>1</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>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>
</tr>
<tr>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

**Byte**

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**

<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>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>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>
</tr>
<tr>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
<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>

**Halfword**

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**

<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>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>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>
</tr>
<tr>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
<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>
Word

SXTW \texttt{<Zd>.D}, \texttt{<Pg>/M}, \texttt{<Zn>.D}

if \texttt{!HaveSVE()} then UNDEFINED;
if \texttt{size} \neq \texttt{11} then UNDEFINED;
integer esize = 8 \ll \texttt{UInt(size)};
integer s_esize = 32;
integer g = \texttt{UInt(Pg)};
integer n = \texttt{UInt(Zn)};
integer d = \texttt{UInt(Zd)};
boolean unsigned = FALSE;

Assembler Symbols

\texttt{<Zd>} Is the name of the destination scalable vector register, encoded in the “Zd” field.
\texttt{<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>

\texttt{<Pg>} Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
\texttt{<Zn>} Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

\texttt{CheckSVEEnabled()};
integer elements = \texttt{VL DIV esize};
bits(PL) mask = \texttt{P[g]};
bits(PL) operand = \texttt{Z[n]};
bits(PL) result = \texttt{Z[d]};
for \( e = 0 \) to elements-1
  bits(esize) element = \texttt{Elem[operand, e, esize]};
  if \texttt{Elem[mask, e, esize]} == \texttt{1} then
    \texttt{Elem[result, e, esize]} = \texttt{Extend(element<s_esize-1:0>, esize, unsigned)};
\texttt{Z[d]} = \texttt{result};

Operational information

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Zm | 0  | 0  | 1  | 0  | 1  | Zn | 0  | Zd |

**SVE2**

TBX <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 = 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

```
<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>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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Odd

```
<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>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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

```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_AA64ZFPR0_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 30 29 28 27 26 25 24 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 1</th>
<th>Zm 0 1 1 1 0 0</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**Even**

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)**

<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 1</th>
<th>Zm 0 0 0 1 1 0</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**Even (quadwords)**

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**

<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 1</th>
<th>Zm 0 1 1 1 0 1</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**Odd**

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)

\[
\begin{array}{cccccccccccccccccc}
\hline
& 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & Zm & 0 & 0 & 0 & 1 & 1 & 1 & Zn & & & & & & & & & & & & & & & & & & & & & & & & & &
\end{array}
\]

Odd (quadwords)

TRN2 \(<Zd>\).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;

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

\(\text{CheckSVEEnabled}()\);
if \(\text{VL} < \text{esize} \times 2\) then UNDEFINED;
integer pairs = \(\text{VL} \div (\text{esize} \times 2)\);
b为其 = \(\text{Z}[\text{n}]\);
b为其 = \(\text{Z}[\text{m}]\);
b为其 = \(\text{Zeros}()\);
for \(p = 0\) to pairs-1
\(\text{Elem}[\text{result}, 2*p+0, \text{esize}] = \text{Elem}[\text{operand1}, 2*p+\text{part}, \text{esize}]\);
\(\text{Elem}[\text{result}, 2*p+1, \text{esize}] = \text{Elem}[\text{operand2}, 2*p+\text{part}, \text{esize}]\);
\(Z[d] = \text{result}\);

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.
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.

```
0 1 0 0 0 1 0 1 size 0  Zm  1 1 1 1 1 1  Zn  Zda
```

UABA `<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);
    boolean unsigned = 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":
  - size `<T>`
    - 00: B
    - 01: H
    - 10: S
    - 11: D

- `<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[da];
for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer element3 = Int(Elem[result, e, esize], unsigned);
    integer res = element3 + Abs(element1 - element2);
    Elem[result, e, esize] = res<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 MOVPREFIX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPREFIX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPREFIX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPREFIX 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.

Signed difference and accumulate (bottom).

Compute signed elements of the difference between the second source vector and the first source vector by subtracting the even-numbered elements of the addend vector from the corresponding elements of the second source vector. The result is destructively added to the corresponding elements of the overlap region of the third 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>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

```
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 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]);
  integer element3 = UInt(Elem[result, e, esize]);
  integer res = element3 + Abs(element1 - element2);
  Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX 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 | 0 | size | 0 | Zm | 1 | 1 | 0 | 0 | 1 | 1 | Zn | Zda |

SVE2

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();
in
teger 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]);
  integer element3 = UInt(Elem[result, e, esize]);
  integer res = element3 + Abs(element1 - element2);
  Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX 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.

SVE

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| size | Zm | 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  |

SVE2

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();
n 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.
**UABDLT**

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.

| 31 30 29 28 27 26 25 24 23 22 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 1 1 1 1 | Zn | Zd |

**SVE2**

UABDLT `<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":  
  |
  | 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**

```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
    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.
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.

### 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

```c
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 ElemP[mask, e, esize] == '0'
    Elem[result, e, esize] = Elem[operand_acc, e, esize];
  else
    integer element1 = UInt(Elem[operand_src, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand_src, 2*e + 1, esize DIV 2]);
    integer element3 = UInt(Elem[operand_acc, e, esize]);
    Elem[result, e, esize] = (element1 + element2 + element3)<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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
UADDBL

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
Zm 0 0 0 0 1 0
Zn 0 0 0 0 1 0
Zd
```

SVE2

UADDBL <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.
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.

<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</td>
</tr>
</tbody>
</table>

SVE2

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 0 1 0 0 | 0 0 0 0 0 1 0 1 | Pg | Zn | Vd |

SVE

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

<table>
<thead>
<tr>
<th>&lt;Dd&gt;</th>
<th>&lt;Pg&gt;</th>
<th>&lt;Zn&gt;</th>
<th>&lt;T&gt;</th>
</tr>
</thead>
</table>
| Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field. | 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. | Is the 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.

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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 1 0 1   | size           | Zm             |
| 0 1 0 0 1 0 0   | Zn             | Zd             |

SVE2

UADDWB <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.
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.

```
| 31 30 29 28 27 26 25 24 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              | Zm                  | Zn                  | Zd                  |
| 0 1 0 0 1 0 1      | 0 1 0 0 1 1         |                    |
```

SVE2

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:
  • The values of the data supplied in any of its registers.
  • 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

<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>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 0 1 0 0 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 16-bit to half-precision

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

<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>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 0 1 0 1 0 1 0 0 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 32-bit to half-precision

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

<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>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
32-bit to single-precision

UCVTF <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 = FPRoundingMode(FPCR);

32-bit to double-precision

UCVTF <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 = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR);

64-bit to half-precision

UCVTF <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;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR);

64-bit to single-precision

UCVTF <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 = FPRoundingMode(FPCR);
64-bit to single-precision

UCVTF <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 = FPRoundingMode(FPCR);

64-bit to double-precision

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 = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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>

SVE

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>”:

<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 = 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is **UNPREDICTABLE**:

- The **MOVPRFX** instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The **MOVPRFX** instructions that can be used with this instruction are as follows:

- An unpredicated **MOVPRFX** instruction.
- A predicated **MOVPRFX** instruction using the same governing predicate register and source element size as 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 |

SVE

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;</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 = 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 = 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

```
| 31 30 29 28 27 26 25 24 23 22 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 0 0 0 0 0 | 1 | Zn | Zda |
```

SVE

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);
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

If PSTATE.DIT is 1:
• The execution time of this instruction is independent of:
  ◦ The values of the data supplied in any of its registers.
  ◦ The values of the NZCV flags.
• The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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


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


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 = \( VL \) DIV esize;
integer elspersegment = 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 elspersegment;
  integer s = segmentbase + index;
  bits(esize) res = \( \text{Elem} \)[operand3, e, esize];
  for i = 0 to 3
    integer element1 = \( \text{UInt} \)(\( \text{Elem} \)[operand1, 4 * e + i, esize DIV 4]);
    integer element2 = \( \text{UInt} \)(\( \text{Elem} \)[operand2, 4 * s + i, esize DIV 4]);
    res = res + element1 * element2;
  \( \text{Elem} \)[result, e, esize] = res;
\( Z[da] \) = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

<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>1</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>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>
</tr>
</tbody>
</table>

| size | Pg | Zm | Zdn |

SVE2

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(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | Pg | Zm | Zdn |

SVE2

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

```
| 31 30 29 28 27 26 25 24 23 22 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 0 0 | Pg  Zm  Zdn |
```

SVE2

UHSUBR <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 = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

SVE

UMAX <$Zdn$.<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];
bite(VL) operand2 = Z[m];
bite(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSiMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 | imm8          | Zdn            |

**SVE**

UMAX <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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX 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.

<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>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>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>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>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>size</th>
<th>Pg</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0</td>
<td>1 0 1 0</td>
<td>1 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

**SVE2**

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.

<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.

**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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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 0 1 0 0 1 0 0 1</td>
</tr>
</tbody>
</table>

**SVE**

UMAXV `<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 = 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 0 1 0 0 | size 0 0 1 0 1 1 0 0 0 | Pg | Zm | Zdn
```

SVE

\text{UMIN} \langle Zdn \rangle . \langle T \rangle , \langle Pg \rangle / M, \langle Zdn \rangle . \langle T \rangle , \langle Zm \rangle . \langle T \rangle

\text{if !HaveSVE()} then UNDEFINED;
integer esize = 8 \ll \text{UInt}(size);
integer g = \text{UInt}(Pg);
integer dn = \text{UInt}(Zdn);
integer m = \text{UInt}(Zm);
boolean unsigned = \text{TRUE};

Assembler Symbols

- \langle Zdn \rangle: Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
- \langle T \rangle: Is the size specifier, encoded in “size”:

<table>
<thead>
<tr>
<th>size</th>
<th>\langle T \rangle</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>

- \langle Pg \rangle: Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
- \langle Zm \rangle: 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};
\text{bits}(\text{PL}) \text{ mask } = P(g);
\text{bits}(\text{VL}) \text{ operand1 } = Z(dn);
\text{bits}(\text{VL}) \text{ operand2 } = Z(m);
\text{bits}(\text{VL}) \text{ result };

for \text{e } = 0 \text{ to elements-1}
  for \text{e1 } = 0 \text{ to esize-1}
    \text{integer element1 } = \text{Int(Elem[operand1, e, esize], unsigned)};
    \text{integer element2 } = \text{Int(Elem[operand2, e, esize], unsigned)};
    \text{if ElemP[mask, e, esize] } = '1' \text{ then}
      \text{integer minimum } = \text{Min(element1, element2)};
      \text{Elem[result, e, esize] } = \text{minimum<esize-1:0>};
    \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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

```
0 0 1 0 0 1 0 1 | size | 1 0 1 0 1 1 1 0 | imm8 | Zdn
```

SVE

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;
bite(VL) operand1 = Z[dn];
bite(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
• An unpredicated MOVPRFX 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  | 0  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

**SVE2**

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**

```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 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>;
    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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  |

SVE

UMINV <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 = 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^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.
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 |

SVE2

UMLALB <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();
n 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]);
    integer element3 = UInt(Elem[result, e, esize]);
    integer res = element3 + element1 * element2;
    Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
  • The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
  • An unpredicated MOVPRFX 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  | i3h | Zm | 1  | 0  | 0  | 1  | i3l | Zn | Zda |
```

#### 32-bit


```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);
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  | 1  | 1  | i2h | Zm | 1  | 0  | 0  | 1  | i2l | Zn | Zda |
```

#### 64-bit

UMLALB `<Zda>.D, <Zn>.S, <Zm>.S[<imm>]`

```plaintext
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 = UInt(Elem[operand1, 2 * e + sel, esize]);
  integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
  integer element3 = UInt(Elem[result, e, 2*esize]);
  integer res = element3 + element1 * element2;
  Elem[result, e, 2*esize] = res<2*esize-1:0>;
Z[da] = result;
```

**Operational information**

If PDST.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

```
<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

UMLALT <Zda>, <T>, <Zn>, <Tb>, <Zm>, <Tb>

if !HaveSVE2() then UNDEFINED;
if size == '00' then UNDEFINED;
type integer esize = 8 << UInt(size);
type integer n = UInt(Zn);
type integer m = UInt(Zm);
type 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]);
  integer element3 = UInt(Elem[result, e, esize]);
  integer res = element3 + element1 * element2;
  Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | i3h| Zm | 1  | 0  | 0  | 1  | i3l| Zn | Zda|

**32-bit**


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  | 0  | 1  | 0  | 0  | 1  | 1  | i2h| Zm | 1  | 0  | 0  | 1  | i2l| Zn | Zda|

**64-bit**


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]);
    integer element3 = UInt(Elem[result, e, 2*esize]);
    integer res = element3 + element1 * element2;
    Elem[result, e, 2*esize] = res<2*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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 |

SVE2

UMLSLB <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 + 0, esize DIV 2]);
  integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
  integer element3 = UInt(Elem[result, e, esize]);
  integer res = element3 - element1 * element2;
  Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
**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

![32-bit encoded vector]


```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);
integer sel = 0;
```

### 64-bit

![64-bit encoded vector]

**UMLSLB** <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```plaintext
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} \text{ DIV} (2 * \text{esize});
integer eltspersegment = 128 \text{ DIV} (2 * \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 \text{ MOD} eltspersegment;
  integer element1 = \texttt{UInt}(\texttt{Elem}[operand1, 2 * e + sel, \text{esize}]);
  integer element2 = \texttt{UInt}(\texttt{Elem}[operand2, 2 * s + index, \text{esize}]);
  integer element3 = \texttt{UInt}(\texttt{Elem}[result, e, 2*\text{esize}]);
  integer res = element3 - element1 * element2;
  \texttt{Elem}[result, e, 2*\text{esize}] = res<2*\text{esize}-1:0>;
\texttt{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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
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 | size | 0 | Zm | 0 1 0 1 1 | Zn | Zda |

SVE2

UMLSLT <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]);
  integer element3 = UInt(Elem[result, e, esize]);
  integer res = element3 - element1 * element2;
  Elem[result, e, esize] = res<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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | i3h | Zm | 1  | 0  | 1  | 1  | i3 | 1  | Zn |   |   |   |   |   |   |   |   |   |   |

32-bit


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  | 1  | 1  | i2h | Zm | 1  | 0  | 1  | 1  | i2 | 1  | Zn |   |   |   |   |   |   |   |   |   |   |

64-bit


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 = \texttt{VL} \DIV (2 * esize);
integer eltspersegment = 128 \DIV (2 * 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, esize]);
    integer element2 = \texttt{UInt}(\texttt{Elem}[operand2, 2 * s + index, esize]);
    integer element3 = \texttt{UInt}(\texttt{Elem}[result, e, 2*esize]);
    integer res = element3 - element1 * element2;
    \texttt{Elem}[result, e, 2*esize] = res<2*esize-1:0>;
\texttt{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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 |
|---------------------------------|---------------------------------|---------------------------------|
| Zm | 1 0 0 1 1 0 | Zn | Zda |

**SVE**


if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;

integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = 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**

```assembly
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**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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   | 0   | 1   | 0   | 0   | 1   | 1   | 0   | 0   | P  | g   | Z   | m   | Zdn |

SVE

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”:

<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 ElemP[mask, e, esize] == '1' then
        integer product = (element1 * element2) >> esize;
        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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

**SVE2**

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.

<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);
   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

<table>
<thead>
<tr>
<th></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>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SVE2

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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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   | 0   | 1   | 0   | 1   | i3h | Zm  | 1   | 1   | 0   | 1   | i3l | 0   | Zn  | Zd  |

#### 32-bit

```assembly

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

<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>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>i2h</td>
<td>Zm</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>i2l</td>
<td>0</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>
</tr>
</tbody>
</table>

#### 64-bit

```assembly
UMULLB <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

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.
**UMULLT (vectors)**

Unsigned multiply long (top).

Multiply 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.

<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>1</td>
<td>size</td>
<td>0</td>
<td>Zm</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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>

**SVE2**

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.
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  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | i3h | Zm | 1  | 1  | 0  | 1 | i3l | Zn |   |   |   |   |   |   |   |   |
```

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  | 0  | 1 | i2l | Zn |   |   |   |   |   |   |   |   |   |
```

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 = 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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

| 31 30 29 28 27 26 25 24 23 22 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 0 0 1 1 0 0 | Pg | Zm | Zdn |

SVE2

UQADD <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 ElemP[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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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".

### SVE

UQADD <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

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

- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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.

### 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   |
- **<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 = \( Int(Elem[operand2, e, esize], unsigned) \);
    \( (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned) \);

\( Z[d] = result; \)

### Operational information

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 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 1 0  imm4 1 1 1 1 1  pattern    Rdn
```

### 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 1 0 0 0 1 0  imm4 1 1 1 1 1  pattern    Rdn
```

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>01110</td>
<td>VL256</td>
</tr>
<tr>
<td>01111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xx00</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>

<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);
```

Operational information

If PSTATE.DIT is 1:

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

```
0 0 0 0 0 0 1 0 0 1 1 1 0 imm4 1 1 1 1 1 pattern Rdn
```

64-bit

```
0 0 0 0 0 0 1 0 0 1 1 1 1 imm4 1 1 1 1 1 pattern Rdn
```

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>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);
```

**Operational information**

If PSTATE.DIT is 1:

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

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | pattern | Zdn |
```

**SVE**

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>000000</td>
<td>POW2</td>
</tr>
<tr>
<td>000001</td>
<td>VL1</td>
</tr>
<tr>
<td>000100</td>
<td>VL2</td>
</tr>
<tr>
<td>000111</td>
<td>VL3</td>
</tr>
<tr>
<td>001000</td>
<td>VL4</td>
</tr>
<tr>
<td>001011</td>
<td>VL5</td>
</tr>
<tr>
<td>001101</td>
<td>VL6</td>
</tr>
<tr>
<td>001111</td>
<td>VL7</td>
</tr>
<tr>
<td>010000</td>
<td>VL8</td>
</tr>
<tr>
<td>010011</td>
<td>VL16</td>
</tr>
<tr>
<td>010100</td>
<td>VL32</td>
</tr>
<tr>
<td>010110</td>
<td>VL64</td>
</tr>
<tr>
<td>011000</td>
<td>VL128</td>
</tr>
<tr>
<td>011011</td>
<td>VL256</td>
</tr>
<tr>
<td>011100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101001</td>
<td>#uimm5</td>
</tr>
<tr>
<td>101010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>110101</td>
<td>#uimm5</td>
</tr>
<tr>
<td>111101</td>
<td>MUL4</td>
</tr>
<tr>
<td>111111</td>
<td>MUL3</td>
</tr>
<tr>
<td></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**

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is **UNPREDICTABLE**:

- The **MOVPRFX** instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The **MOVPRFX** instructions that can be used with this instruction are as follows:

- An unpredicated **MOVPRFX** instruction.
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

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

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":

UQDECH (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>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);
```

**Operational information**

If PSTATE.DIT is 1:

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

Unsigned 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 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 | imm4 | 1 1 0 0 1 1 | pattern | Zdn
```

**SVE**

UQDECH <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>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>10111</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11001</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</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

```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

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  | 1  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

32-bit

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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |

64-bit

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

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);

Operational information

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

```
0 0 1 0 0 1 0 1 | size 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0
```

SVE

```
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

If PSTATE.DIT is 1:

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

- The response of this instruction to 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 MOVEPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVEPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVEPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVEPRFX 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**

31 30 29 28 27 26 25 24 23 22 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 imm4 1 1 1 1 1 pattern Rdn

**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 1 0 1 1 imm4 1 1 1 1 1 pattern Rdn

**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":

```asm
UQDECW <Wdn>{, <pattern>{, MUL #<imm>}}
```

```asm
UQDECW <Xdn>{, <pattern>{, MUL #<imm>}}
```
<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>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);
bites(ssize) operand1 = X[dn];
bites(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn] = Extend(result, 64, unsigned);

Operational information

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

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

\[
\begin{array}{ccccccccccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \text{imm4} & 1 & 1 & 0 & 0 & 1 & 1 & \text{pattern} & \text{Zdn}
\end{array}
\]

SVE

\[
\text{UQDECW} \ <\text{Zdn}>.S\{, \ <\text{pattern}>\{, \ MUL \ #\text{<imm>}}\}
\]

if !\text{HaveSVE}() then UNDEFINED;
integer esize = 32;
integer dn = \text{UInt}(\text{Zdn});
bits(5) pat = \text{pattern};
integer imm = \text{UInt}(\text{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>01110</td>
<td>VL128</td>
</tr>
<tr>
<td>01111</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>1x000</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 = \texttt{VL} \texttt{DIV} \texttt{esize};
integer count = \texttt{DecodePredCount}(\texttt{pat}, \texttt{esize});
bits(\texttt{VL}) operand1 = \texttt{Z}[\texttt{dn}];
bits(\texttt{VL}) result;

\textbf{for e = 0 to elements-1} \\
\hspace{1em} integer element1 = \texttt{Int(Elem[operand1, e, esize], unsigned)};
\hspace{1em} (\texttt{Elem[result, e, esize]}, \texttt{-}) = \texttt{SatQ}(element1 - (count * \texttt{imm}), \texttt{esize, unsigned});
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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
UQINCB

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  | 1  | 0  | 0  | 0  | 1  | 0  | imm4 | 1  | 1  | 1  | 1  | 0  | 1  | pattern | Rdn |

32-bit

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  | 1  | 0  | 0  | 0  | 1  | 1  | imm4 | 1  | 1  | 1  | 1  | 0  | 1  | pattern | Rdn |

64-bit

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>&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>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);
```

Operational information

If PSTATE.DIT is 1:

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

![32-bit encoding](image)

```assembly
UQINCD <Wdn>{, <pattern>{, MUL #<imm>}}
```

```assembly
def if !HaveSVE() then UNDEFINED;
def integer esize = 64;
def integer dn = UInt(Rdn);
def bits(5) pat = pattern;
def integer imm = UInt(imm4) + 1;
def boolean unsigned = TRUE;
def integer ssize = 32;
```

#### 64-bit

![64-bit encoding](image)

```assembly
UQINCD <Xdn>{, <pattern>{, MUL #<imm>}}
```

```assembly
def if !HaveSVE() then UNDEFINED;
def integer esize = 64;
def integer dn = UInt(Rdn);
def bits(5) pat = pattern;
def integer imm = UInt(imm4) + 1;
def boolean unsigned = TRUE;
def 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":

---

---
<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);
```

**Operational information**

If PSTATE.DIT is 1:

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

```
31 30 29 28 27 26 25 24 23 22 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 | 1 |
| pattern | Zdn |
```

**SVE**

```
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>#imm5</td>
</tr>
<tr>
<td>1011x</td>
<td>#imm5</td>
</tr>
<tr>
<td>10110</td>
<td>#imm5</td>
</tr>
<tr>
<td>10101</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

```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

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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  | 1  | 1  | 0  | imm4| 1  | 1  | 1  | 1  | 0  | 1  | pattern | Rdn |

32-bit

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  | 1  | 1  | 1  | imm4| 1  | 1  | 1  | 1  | 0  | 1  | pattern | Rdn |

64-bit

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":

<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>1x010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>1xxx0</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);
```

**Operational information**

If PSTATE.DIT is 1:

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

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

<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</td>
</tr>
</tbody>
</table>

SVE

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>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>1x0x0</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

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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

<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>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>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></td>
</tr>
</tbody>
</table>

Pm | Rdn

32-bit

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

<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>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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pm | Rdn

64-bit

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>

UQINCP (scalar)
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);

Operational information

If PSTATE.DIT is 1:

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

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

| 31 30 29 28 27 26 25 24 23 22 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 | size 1 0 1 0 | 0 1 0 0 0 | 0 0 | Pm | Zdn |

SVE

UQINCP <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

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | 1  | 1  | 1  | 0  | 1  | imm4 | 1 | 1 | 1 | 0 | 1  | pattern | Rdn |

#### 32-bit

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

| 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  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  | 1  | imm4 | 1 | 1 | 1 | 1 | 0 | 1  | pattern | Rdn |

#### 64-bit

UQINCW <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":

<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);
```

**Operational information**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
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 0 1 0 0 | 1 | 0 | 1 | 0 | imm4 | 1 | 1 | 0 | 0 | 0 | 1 | pattern | Zdn |
```

SVE

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>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>01111</td>
<td>VL256</td>
</tr>
<tr>
<td>01100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10001</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10100</td>
<td>#uimm5</td>
</tr>
<tr>
<td>10110</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11000</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11010</td>
<td>#uimm5</td>
</tr>
<tr>
<td>11100</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

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX 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.

```
| 31 30 29 28 27 26 25 24 23 22 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 1 0 1 1 0 0 | Pg Zm Zdn           |
```

**SVE2**

UQRSHL <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Zm>.<T>

```java
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**

```java
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 ElemP[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**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

![Vector Register](image)

SVE2

UQRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

if !HaveSVE2() then UNDEFINED;
i

integer esize = \(8 \ll \text{UInt}(\text{size})\);
i

integer g = \text{UInt}(\text{Pg});
i

integer m = \text{UInt}(\text{Zm});
i

integer dn = \text{UInt}(\text{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>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

\[
\text{CheckSVEEnabled}();
\]

\[
i
\text{integer elements} = \text{VL} \div \text{esize};
\]

\[
i
\text{bits(PL) mask} = \text{P}[g];
\]

\[
i
\text{bits(VL) operand1} = \text{Z}[m];
\]

\[
i
\text{bits(VL) operand2} = \text{Z}[dn];
\]

\[
i
\text{bits(VL) result};
\]

for \(e = 0 \) to elements-1

\[
\text{integer element} = \text{UInt(Elem[operand1, e, esize])};
\]

\[
\text{integer shift} = \text{ShiftSat(SInt(Elem[operand2, e, esize]), esize)};
\]

\[
\text{integer round const} = 1 \ll (-\text{shift} - 1); // 0 for left shift, }2^{(n-1)}\text{ for right shift}
\]

if \(\text{Elem}[\text{mask, e, esize}] == '1'\) then

\[
\text{integer res} = (\text{element} + \text{round const}) \ll \text{shift};
\]

\[\text{Elem[result, e, esize]} = \text{UnsignedSat(res, esize)};\]

else

\[
\text{Elem[result, e, esize]} = \text{Elem[operand2, e, esize]};
\]

\[
\text{Z[dn]} = \text{result};
\]

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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 0 tszh 1 tszl imm3 0 0 1 1 1 0 Zn Zd</td>
</tr>
</tbody>
</table>

#### SVE2

\[\text{UQRSHRNB } <\text{Zd}>,<\text{T}>,<\text{Zn}>,<\text{Tb}>, #<\text{const}>\]

```plaintext
if !\text{HaveSVE2()} \text{ 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 = \text{UInt}(\text{Zn});
integer d = \text{UInt}(\text{Zd});
integer shift = (2 * esize) - \text{UInt}(\text{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”:
  ```plaintext
  tszh tszl  <T>
  0 00  RESERVED
  0 01  B
  0 1x  H
  1 xx  S
  ```
- **<Zn>** Is the name of the source scalable vector register, encoded in the "Zn" field.
- **<Tb>** Is the size specifier, encoded in “tszh:tszl”:
  ```plaintext
  tszh tszl  <Tb>
  0 00  RESERVED
  0 01  H
  0 1x  S
  1 xx  D
  ```
- **<const>** Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3”.

#### Operation

```plaintext
\text{CheckSVEEnabled();}
integer elements = \text{VL} \div (2 * esize);
bits(\text{VL}) operand = \text{Z}[n];
bits(\text{VL}) result;
integer round const = 1 \ll (shift-1);
for e = 0 to elements-1
  bits(2*esize) element = \text{Elem}[operand, e, 2*esize];
  integer res = (\text{UInt}(element) + round const) \gg shift;
  \text{Elem}[result, 2*e + 0, esize] = \text{UnsignedSat}(res, esize);
  \text{Elem}[result, 2*e + 1, esize] = \text{Zeros}();
\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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | Zn | Zd |

SVE2

UQRSHRNT <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 "tszh: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] = UnsignedSat(res, 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.
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.

![SVE2](image)

**Assemble 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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**UQSHL (vectors)**

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.

### SVE2

UQSHL \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>\)

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);  
    if ElemP[mask, e, esize] == '1' then
        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

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 \texttt{MOVPRFX} instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is \texttt{UNPREDICTABLE}:

- 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.

The \texttt{MOVPRFX} instructions that can be used with this instruction are as follows:

- An unpredicated \texttt{MOVPRFX} instruction.
- A predicated \texttt{MOVPRFX} instruction using the same governing predicate register and source element size as this instruction.
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 to \(2^N - 1\). 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**

```c
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 ElemP[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**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09 rc2_1, sve v2019-09 rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
UQSHRNB

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 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.

| 31  | 30  | 29  | 28  | 27  | 26  | 25  | 24  | 23  | 22  | 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   | 0   | 1   | 1   | 0   | 0   | Zn  | 0   | 1   | 1   | 0   |

SVE2

UQSHRNB <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;

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;
Operational information

If PSTATE.DIT is 1:

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

- The response of this instruction to 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 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.

SVE2

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 = 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] = UnsignedSat(res, 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.
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.

| 31 30 29 28 27 26 25 24 23 22 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 | 0 | 1 | 1 | 1 | 0 | 0 | | Pg | Zm | Zdn |

SVE2

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 ElemP[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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 "#<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".

```
0 0 0 1 0 0 1 1 | size 1 0 0 1 1 1 1 1 | sh | imm8 | Zdn
```

SVE

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

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

- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is \texttt{UNPREDICTABLE}:

- 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.

The \texttt{MOVPRFX} instructions that can be used with this instruction are as follows:

- An unpredicated \texttt{MOVPRFX} 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.

```
|   31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  | Zm | 0  | 0  | 0  | 1  | 1  | 1  | Zn | 1  | Zd |
```

**SVE**

UQSUB `<Zd>.<T>, <Zn>.<T>, <Zm>.<T>`

```plaintext
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><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
  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;
```

**Operational information**

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

**SVE2**

UQSUBR \(<Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>\)

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 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] = 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 0 1 0 1 0 tszh:tszl 0 0 0 0 1 0 0 1 Zn 1 Zd
```

**SVE2**

**UQXTNB <Zd>.<T>, <Zn>.<Tb>**

```plaintext
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 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;
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;

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

\[
\begin{array}{cccccccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & Zn & Zd \\
\end{array}
\]

SVE2

if !HaveSVE2() then UNDEFINED;

\[
tsize = tszh:tszl;
\]

\[
\begin{array}{|c|c|}
\hline
\text{tszh} & \text{tszl} \\
\hline
0 & 00 & \text{RESERVED} \\
0 & 01 & B \\
0 & 10 & H \\
x & 11 & \text{RESERVED} \\
1 & 00 & S \\
1 & 01 & \text{RESERVED} \\
1 & 10 & \text{RESERVED} \\
\hline
\end{array}
\]

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”:

\(<Zn>\) Is the name of the first source scalable vector register, encoded in the “Zn” field.

\(<Tb>\) Is the size specifier, encoded in “tszh:tszl”:

Operation

\(\text{CheckSVEEnabled}()\);

\[\text{integer elements} = \frac{\text{VL}}{\text{esize}};\]

\[\text{bits(VL)} \text{ operand1} = Z[n];\]

\[\text{bits(VL)} \text{ result} = Z[d];\]

\[\text{integer halfesize} = \frac{\text{esize}}{2};\]

for \(e = 0\) to elements-1

\[\text{integer element1} = \text{UInt}([\text{Elem}[\text{operand1}, e, \text{esize}]]) ;\]

\[\text{bits(halfesize)} \text{ res} = \text{UnsignedSat}([\text{element1}, \text{halfesize}] , \text{esize}) ;\]

\[\text{Elem}[\text{result}, 2*e + 1, \text{halfesize}] = \text{res};\]

\[Z[d] = \text{result};\]
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.
**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 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>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>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>
<td></td>
</tr>
</tbody>
</table>

**SVE2**

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 ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = UnsignedRecipEstimate(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
A predicated `MOVPRFX` instruction using the same governing predicate register and source element size as 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  | 0  | 1  | 0  | 0  | size| 0  | 1  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | Pg | Zm | Zdn |

**SVE2**

URHADD <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 = 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**

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 0  |
The values of the NZCV 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
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.

31 30 29 28 27 26 25 24 23 22 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 1 1 1 1 0 0

Pg | Zm | Zdn

SVE2

URSHLR <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);
   integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
   if ElemP[mask, e, esize] == '1' then
      integer res = (element + round_const) << shift;
   else
      Elem[result, e, esize] = res<esize-1:0>;
   Elem[result, e, esize] = Elem[operand2, 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
**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 1 0 0 | tszh | 0 0 | 1 1 | 0 1 | 1 | 0 0 | Pg | tszl | imm3 | Zdn |

**SVE2**

**URSHR** <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</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;

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

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.
**URSQRT**

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  | 1  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 1  | Pg | Zn | Zd |

**SVE2**

**URSQRT** <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 ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = UnsignedRSqrtEstimate(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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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.

<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 tszh 0 tszl imm3 1 1 1 0 1 1 Zn Zda</td>
</tr>
</tbody>
</table>

**SVE2**

URSRA `<Zda>..<T>`, `<Zn>..<T>`, `#<const>`

```java
if !HaveSVE2() then UNDEFINED;
b设立 {} 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**

```java
CheckSVEEnabled();
integer elements = VL DIV esize;
b设立 {} operand1 = Z[n];
b设立 {} operand2 = Z[da];
b设立 {} 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**

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
• The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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  | Zm | 0  | 1  | 1  | 1  | 0  | Zn | Zda |

SVE


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

If PSTATE.DIT is 1:
  • The execution time of this instruction is independent of:
    ◦ The values of the data supplied in any of its registers.
    ◦ The values of the NZCV flags.
  • The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
  • The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand of this instruction.

The `MOVPRFX` instructions that can be used with this instruction are as follows:

• An unpredicated `MOVPRFX` 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 i2 Zm 0 0 0 1 1 0 Zn Zda

SVE


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

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

If PSTATE.DIT is 1:

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

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

This instruction might be immediately preceded in program order by a MOVPRFX instruction that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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 0 tszh</td>
</tr>
</tbody>
</table>

SVE2

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</td>
<td>00</td>
<td>RESERVE</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>

<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>RESERVE</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);
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.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & tszh & 0 & tszl & imm3 & 1 & 0 & 1 & 0 & 1 & 1 & Zn & & Zd & & \end{array}
\]

SVE2

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</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>

<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>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);
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.

SVE


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

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

If PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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  0  1  0  1  1  0  1  1  0  0  Pg  Zm  Zdn
```

**SVE**

**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”:

<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] = UnsignedSat(UInt(element1) + SInt(element2), esize);
  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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX instruction.
- A predicated MOVPRFX instruction using the same governing predicate register and source element size as 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 |

**SVE2**

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>tzh</th>
<th>tzl</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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

- An unpredicated MOVPRFX 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.

```
31 30 29 28 27 26 25 24 23 22 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 0 1 1 0 Zn Zd
```

SVE2

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.

|   |   |   |   |   |   |   |   |   |   |   | size |   | Zm |   | Zn |   | Zd |
|---|---|---|---|---|---|---|---|---|---|---|-----|---|----|---|----|---|
| 31| 30| 29| 28| 27| 26| 25| 24| 23| 22| 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 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |

SVE2

USUBLT <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.
**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.

![Vector Register Format](image)

**SVE2**

USUBWB \(<Zd>..<T>, <Zn>..<T>, <Zm>..<Tb>\)

if \(!\text{HaveSVE2}()\) 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**

\(\text{CheckSVEEnabled}()\);
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 = UInt(\text{Elem}[operand1, e, esize]);
  integer element2 = UInt(\text{Elem}[operand2, 2*e + 0, esize \div 2]);
  \text{Elem}[result, e, esize] = (element1 - element2)<\text{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 response of this instruction to 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.

```
31 30 29 28 27 26 25 24 23 22 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 1 Zn Zd
```

SVE2

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>&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 response of this instruction to 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

<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>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>1 1 0 0</td>
<td>1 1 0 0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

High half

UUNPKHI <Zd>.<T>, <Zn>.<Tb>

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

<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>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 1 0 1</td>
<td>1 1 0 0</td>
<td>1 0 0 0 1 1 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Low half

UUNPKLO <Zd>.<T>, <Zn>.<Tb>

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

\textbf{CheckSVEEnabled}();
integer elements = \text{VL} \div \text{esize};
integer hsize = esize \div 2;
bits(\text{VL}) operand = Z[n];
bits(\text{VL}) result;

for e = 0 to elements-1
  bits(hsize) element = if hi then \text{Elem}[operand, e + elements, hsize] else \text{Elem}[operand, e, hsize];
  \text{Elem}[result, e, esize] = \text{Extend}(element, esize, unsigned);

\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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 30 29 28 27 26 25 24 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 0 1 0 0 1 1 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**UXTB <Zd>.<T>, <Pg>M, <Zn>.<T>**

```c
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 30 29 28 27 26 25 24 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 0 1 0 0 1 1 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>

**UXTH <Zd>.<T>, <Pg>M, <Zn>.<T>**

```c
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 30 29 28 27 26 25 24 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 0 1 0 1 1 0 1</th>
<th>Pg</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
</table>
Word

\[ \text{UXTW } <Zd>, <Pg>/M, <Zn>.D \]

if \(!\text{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

\(\text{CheckSVEEnabled}();\)
integer elements = \(\text{VL} \div \text{esize}\);
bits(PL) mask = \(P[g]\);
bits(VL) operand = \(Z[n]\);
bits(VL) result = \(Z[d]\);
for e = 0 to elements-1
   bits(esize) element = \(\text{Elem}[\text{operand}, e, \text{esize}]\);
   if \(\text{Elem}[\text{mask}, e, \text{esize}] == '1'\) then
      \(\text{Elem}[\text{result}, e, \text{esize}] = \text{Extend}(\text{element<s_esize-1:0>}, \text{esize}, \text{unsigned});\)
\(Z[d] = \text{result};\)

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 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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:
- An unpredicated MOVPRFX instruction.
• A predicated MOVPRFX instruction using the same governing predicate register and source element size as this instruction.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
**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 1 0 1 0 1 | size | 1 0 | Pm | 0 1 0 0 1 0 0 | Pn | 0 | Pd |

**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 0 1 1 0 | Pn | 0 | 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 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) 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 | Zm | 0 1 1 0 1 0 | Zn | Zd |

Even

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 | Zm | 0 0 0 0 1 0 | Zn | Zd |

Even (quadwords)

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 | Zm | 0 1 1 0 1 1 | Zn | Zd |

Odd

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)

```
0 0 0 0 0 1 0 1 | Zm 0 0 0 0 1 1 | Zn 0 0 0 0 1 1 | Zd
```

Odd (quadwords)

```
UZP2 <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;
```

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.

\[
\begin{array}{cccccccccccccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & \text{size} & 1 & \text{Rm} & 0 & 0 & 0 & \text{sf} & 0 & 0 & \text{Rn} & 0 & \text{Pd} \\
\end{array}
\]

SVE2

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

<\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>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{R}> Is a width specifier, encoded in “sf”:

<table>
<thead>
<tr>
<th>sf</th>
<th>&lt;\text{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

`CheckSVEEnabled();`
integer elements = `VL` DIV `esize`;
bits(`PL`) mask = `Ones(`PL`);
bits(`rsz`) operand1 = `X[n]`;
bits(`rsz`) 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 \texttt{FIRST} (N), \texttt{NONE} (Z), \texttt{!LAST} (C) condition flags based on the predicate result, and the V flag to zero.

\begin{center}
\begin{tabular}{cccccccccccccccc}
  0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & | & \texttt{size} & | & 1 & | & Rm & | & 0 & 0 & 0 & | & sf & | & 0 & 0 & | & Rn & | & 1 & | & Pd
\end{tabular}
\end{center}

\textbf{SVE2}

\texttt{WHILEGT} <\texttt{Pd}>, <\texttt{T}>, <\texttt{R}<\texttt{n}>>, <\texttt{R}<\texttt{m}>>

if !\texttt{HaveSVE2()} then UNDEFINED;
integer esize = 8 \ll \texttt{UInt}(\texttt{size});
integer rsize = 32 \ll \texttt{UInt}(\texttt{sf});
n = \texttt{UInt}(\texttt{Rn});
m = \texttt{UInt}(\texttt{Rm});
d = \texttt{UInt}(\texttt{Pd});
boolean unsigned = FALSE;
\texttt{SVECmp} \texttt{op} = \texttt{Cmp_GT};

\textbf{Assembler Symbols}

\texttt{<Pd>} Is the name of the destination scalable predicate register, encoded in the "Pd" field.

\texttt{<T>} Is the size specifier, encoded in "\texttt{size}":

\begin{center}
\begin{tabular}{c|c}
  \texttt{size} & \texttt{<T>} \\
  00 & B \\
  01 & H \\
  10 & S \\
  11 & D
\end{tabular}
\end{center}

\texttt{<R>} Is a width specifier, encoded in "\texttt{sf}":

\begin{center}
\begin{tabular}{c|c}
  \texttt{sf} & \texttt{<R>} \\
  0 & W \\
  1 & X
\end{tabular}
\end{center}

\texttt{<n>} Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "\texttt{Rn}" field.

\texttt{<m>} Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "\texttt{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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

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

```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.
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.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 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  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

SVE2

WHILEHS <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_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>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

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.

**SVE**

WHILELE <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 = FALSE;
SVECmp op = Cmp_LE;

**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

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 | size | 1 | Rm | 0 0 0 | sf | 1 | 1 | Rn | 0 | Pd
```

SVE

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.

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.
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.

31 30 29 28 27 26 25 24 23 22 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 | 1 1 | Rn | 1 | Pd |

SVE

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

<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.
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.

```
      31  30  29  28  27  26  25  24  23  22  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  1  Rn  0  Pd
```

SVE

WHILELT <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 = FALSE;
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

```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.
WHILERW

While free of read-after-write conflicts.

This instruction checks two addresses for a conflict or overlap between address ranges of the form \([addr, addr + \text{VL} ÷ 8]\), where \(\text{VL}\) 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.

```
\[
\begin{array}{ccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & \text{size} & 1 & \text{Rm} & 0 & 0 & 1 & 1 & 0 & 0 & \text{Rn} & 1 & \text{Pd}
\end{array}
\]
```

SVE2

WHILERW \(<\text{Pd}>.\langle T\rangle, \langle Xn\rangle, \langle Xm\rangle>

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

\(<\text{Pd}>\) Is the name of the destination scalable predicate register, encoded in the "Pd" field.
\(<\text{T}>\) Is the size specifier, encoded in “size”:

\[
\begin{array}{c|c}
\text{size} & \langle \text{T} \rangle \\
00 & B \\
01 & H \\
10 & S \\
11 & D \\
\end{array}
\]

\(<\text{Xn}>\) Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.
\(<\text{Xm}>\) Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

Operation

\text{CheckSVEEnabled();}
integer elements = \text{VL} \text{ DIV esize};
\text{bits(PL)} mask = \text{Ones(PL)};
\text{bits(64) src1} = \text{X}[n];
\text{bits(64) src2} = \text{X}[m];
integer operand1 = \text{UInt(src1)};
integer operand2 = \text{UInt(src2)};
\text{bits(PL)} result;

integer diff = \text{Abs}(\text{operand2} - \text{operand1}) \text{ DIV (esize DIV 8)};
for e = 0 to elements-1
    if diff == 0 || e < diff then
        \text{ElemP}[\text{result}, e, esize] = '1';
    else
        \text{ElemP}[\text{result}, e, esize] = '0';
\end{if}
PSTATE.<N,Z,C,V> = \text{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.
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 \([addr, addr+VL+8)\), 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.

![](image)

SVE2

WHILEWR \(<Pd>\), \(<T>\), \(<Xn>\), \(<Xm>\)

\[
\text{if } \text{!HaveSVE2}() \text{ then UNDEFINED;}
\]

\[
\text{integer esize} = 8 \ll \text{UInt}(\text{size});
\]

\[
\text{integer } n = \text{UInt}(Rn);
\]

\[
\text{integer } m = \text{UInt}(Rm);
\]

\[
\text{integer } d = \text{UInt}(Pd);
\]

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

\[\text{CheckSVEEnabled}();\]

\[
\text{integer elements} = \text{VL} \div \text{esize};
\]

\[\text{bits(PL)} \text{ mask} = \text{Ones(PL)};\]

\[\text{bits(64) src1} = X[n];\]

\[\text{bits(64) src2} = X[m];\]

\[\text{integer operand1} = \text{UInt}(\text{src1});\]

\[\text{integer operand2} = \text{UInt}(\text{src2});\]

\[\text{bits(PL)} \text{ result};\]

\[
\text{integer diff} = (\text{operand2} - \text{operand1}) \div (\text{esize} \div 8);
\]

\[
\text{for } e = 0 \text{ to elements-1 }
\]

\[
\text{if diff} \leq 0 \text{ || } e < \text{diff then}
\]

\[
\text{ElemP}[\text{result}, e, \text{esize}] = '1';
\]

\[
\text{else}
\]

\[
\text{ElemP}[\text{result}, e, \text{esize}] = '0';
\]

\[\text{PSTATE.}<N,Z,C,V> = \text{PredTest}(\text{mask}, \text{result}, \text{esize});\]

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

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.
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 0 1 0 1 0 0 1 0 0 1 0 0 0 0 Pn | 0 | 0 | 0 | 0
```

SVE

```
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 1 0 0 tszh | 1 tszl | imm3 | 0 0 1 1 0 1 | Zm | Zdn |

SVE2

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 that conforms to all of the following requirements, otherwise the behavior of either or both instructions is UNPREDICTABLE:

• The MOVPRFX instruction must specify the same destination register as this instruction.
• The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

The MOVPRFX instructions that can be used with this instruction are as follows:

• An unpredicated MOVPRFX 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

<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</th>
<th>Pm</th>
<th>0 1 0 0 1 0</th>
<th>Pn</th>
<th>0</th>
<th>Pd</th>
</tr>
</thead>
</table>

High halves

ZIP2 <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;

Low halves

<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</th>
<th>Pm</th>
<th>0 1 0 0 0 0</th>
<th>Pn</th>
<th>0</th>
<th>Pd</th>
</tr>
</thead>
</table>

Low halves

ZIP1 <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;

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;I&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

```c
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_AA64SFPR0_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

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)

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

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)

|   | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |   | Zm | 0 | 0 | 0 | 0 | 0 |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 |   | Zn |   |   |   |   |   |   |   |   |   |   |

ZIP1 <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;

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();
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
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, 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.
### Top-level encodings for A64

**Decode fields**

<table>
<thead>
<tr>
<th>op0</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0010</td>
<td>SVE encodings</td>
</tr>
<tr>
<td>0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100x</td>
<td>Data Processing -- Immediate</td>
</tr>
<tr>
<td>101x</td>
<td>Branches, Exception Generating and System instructions</td>
</tr>
<tr>
<td>x1x0</td>
<td>Loads and Stores</td>
</tr>
<tr>
<td>x101</td>
<td>Data Processing -- Register</td>
</tr>
<tr>
<td>x111</td>
<td>Data Processing -- Scalar Floating-Point and Advanced SIMD</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>0000</td>
<td>UDF</td>
</tr>
<tr>
<td>! = 0000</td>
<td>UNALLOCATED</td>
<td>UNALLOCATED</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>000xxx</td>
<td>SVE Integer Binary Arithmetic - Predicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>001xxx</td>
<td>SVE Integer Reduction</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>100xxx</td>
<td>SVE Bitwise Shift - Predicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>0xxxx</td>
<td>101xxx</td>
<td>SVE Integer Unary Arithmetic - Predicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>000xxx</td>
<td>SVE integer add/subtract vectors (unpredicated)</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>001xxx</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>011xxx</td>
<td>SVE2 Integer Multiply - Unpredicated</td>
</tr>
<tr>
<td>000</td>
<td>0x</td>
<td>1xxxx</td>
<td>100xxx</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>Bit Pattern</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 0x 1xxxx 11xxxx</td>
<td>SVE Element Count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 1x 00xxx</td>
<td>SVE Bitwise Immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 1x 01xxxx</td>
<td>SVE Integer Wide Immediate - Predicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 1x 1xxxx 001xxxx</td>
<td>SVE Permute Vector - Unpredicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 1x 1xxxx 010xxxx</td>
<td>SVE Permute Predicate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 1x 1xxxx 011xxxx</td>
<td>SVE permute vector elements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 1x 1xxxx 10xxxx</td>
<td>SVE Permute Vector - Predicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 1x 1xxxx 11xxxx</td>
<td>SEL (vectors)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 10 1xxxx 000xxxx</td>
<td>SVE Permute Vector - Extract</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 11 1xxxx 000xxxx</td>
<td>SVE permute vector segments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 0x 0xxxxx</td>
<td>SVE Integer Compare - Vectors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 0x 1xxxx</td>
<td>SVE integer compare with unsigned immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 0xxxxx x0xxxx</td>
<td>SVE integer compare with signed immediate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 00xxx 01xxxx</td>
<td>SVE predicate logical operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 00xxx 11xxxx</td>
<td>SVE Propagate Break</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 01xxx 01xxxx</td>
<td>SVE Partition Break</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 01xxx 11xxxx</td>
<td>SVE Predicate Misc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 1xxxx 00xxxx</td>
<td>SVE Integer Compare - Scalars</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 1xxxx 01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 1xxxx 11xxxx</td>
<td>SVE Integer Wide Immediate - Unpredicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 100xx 10xxxx</td>
<td>SVE predicate count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 101xx 1000xx</td>
<td>SVE Inc/Dec by Predicate Count</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 101xx 1001xx</td>
<td>SVE Write FFR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 101xx 101xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1x 11xxx 10xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 0x 0xxxx 0xxxxx</td>
<td>SVE Integer Multiply-Add - Unpredicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 0x 0xxxx 10xxxx</td>
<td>SVE2 Integer - Predicated</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 0x 0xxxx 11xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 0x 1xxxx</td>
<td>SVE Multiply - Indexed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1x 0xxxx 0xxxxx</td>
<td>SVE2 Widening Integer Arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1x 0xxxx 10xxxx</td>
<td>SVE Misc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1x 0xxxxx 11xxxx</td>
<td>SVE2 Accumulate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1x 1xxxx 0xxxxx</td>
<td>SVE2 Narrowing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1x 1xxxx 100xx</td>
<td>SVE2 character match</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1x 1xxxx 101xx</td>
<td>SVE2 Histogram Computation - Segment</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1x 1xxxx 110xxxx</td>
<td>HISTCNT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 1x 1xxxx 111xxxx</td>
<td>SVE2 Crypto Extensions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 0xxxx 0xxxxx</td>
<td>FCMLA (vectors)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 00x1x 1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 00000 100xx</td>
<td>FCADD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 00000 101xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 00000 11xxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 00001 1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 0010x 100xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 0010x 101xx</td>
<td>SVE floating-point convert precision odd elements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 010xx 100xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 010xx 101xx</td>
<td>SVE2 floating-point pairwise operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 010xx 110xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0x 010xx 111xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SVE Integer Multiply-Add - Predicated

These instructions are under SVE encodings.

| 011 | 0x | 011xx | 1xxxxx | UNALLOCATED |
| 011 | 0x | 1xxxxx | x0x01x | UNALLOCATED |
| 011 | 0x | 1xxxxx | 00000x | SVE floating-point multiply-add (indexed) |
| 011 | 0x | 1xxxxx | 0001xx | SVE floating-point complex multiply-add (indexed) |
| 011 | 0x | 1xxxxx | 001000 | SVE floating-point multiply (indexed) |
| 011 | 0x | 1xxxxx | 001001 | UNALLOCATED |
| 011 | 0x | 1xxxxx | 0011xx | UNALLOCATED |
| 011 | 0x | 1xxxxx | 0100xx | SVE floating-point multiply-accumulate |
| 011 | 0x | 1xxxxx | 0101xx | SVE floating-point matrix multiply accumulate |
| 011 | 0x | 1xxxxx | 110000 | UNALLOCATED |
| 011 | 0x | 1xxxxx | 110001 | SVE floating point matrix multiply accumulate |
| 011 | 0x | 1xxxxx | 1110xx | UNALLOCATED |
| 011 | 0x | 1xxxxx | 1111xx | UNALLOCATED |
| 011 | 1x | 0xxxxx | x1xxxx | SVE floating-point compare vectors |
| 011 | 1x | 0xxxxx | 0000xx | SVE floating-point arithmetic (unpredicated) |
| 011 | 1x | 0xxxxx | 100xxx | SVE Floating Point Arithmetic - Predicated |
| 011 | 1x | 0xxxxx | 101xxx | SVE Floating Point Unary Operations - Predicated |
| 011 | 1x | 000xx | 001xxx | SVE floating-point recursive reduction |
| 011 | 1x | 001xx | 0010xx | UNALLOCATED |
| 011 | 1x | 001xx | 0011xx | SVE Floating Point Unary Operations - Unpredicated |
| 011 | 1x | 010xx | 001xxx | SVE Floating Point Compare - with Zero |
| 011 | 1x | 011xx | 001xxx | SVE floating-point serial reduction (predicated) |
| 011 | 1x | 1xxxxx | SVE Floating Point Multiply-Add |
| 100 | | | | SVE Memory - 32-bit Gather and Unsized Contiguous |
| 101 | | | | SVE Memory - Contiguous Load |
| 110 | | | | SVE Memory - 64-bit Gather |
| 111 | 0x0xxx | 0x0xxx | SVE Memory - Contiguous Store and Unsized Contiguous |
| 111 | 0x1xxx | 0x1xxx | SVE Memory - Non-temporal and Multi-register Store |
| 111 | 1x0xxx | 1x0xxx | SVE Memory - Scatter with Optional Sign Extend |
| 111 | 101xxx | 101xxx | SVE Memory - Scatter |
| 111 | 111xxx | 111xxx | SVE Memory - Contiguous Store with Immediate Offset |

SVE Integer Multiply-Add - Predicated

These instructions are under SVE encodings.

| 00000100 | 0 | op0 | 1 |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SVE integer multiply-accumulate writing addend (predicated)</td>
</tr>
<tr>
<td>1</td>
<td>SVE integer multiply-add writing multiplicand (predicated)</td>
</tr>
</tbody>
</table>

SVE integer multiply-accumulate writing addend (predicated)

These instructions are under SVE Integer Multiply-Add - Predicated.
### SVE integer multiply-add writing multiplicand (predicated)

These instructions are under [SVE Integer Multiply-Add - Predicated](#).

<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 Binary Arithmetic - Predicated

These instructions are under [SVE encodings](#).

#### SVE integer add/subtract 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>00x</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 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>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>
### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>SMAX (vectors)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>UMAX (vectors)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>SMIN (vectors)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>UMIN (vectors)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>SABD</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>UABD</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE integer multiply vectors (predicated)

These instructions are under **SVE Integer Binary Arithmetic - 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  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Zm | Zdn |

### Decode fields

<table>
<thead>
<tr>
<th>H</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>MUL (vectors, predicated)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>SMULH (predicated)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>UMULH (predicated)</td>
</tr>
</tbody>
</table>

### SVE integer divide vectors (predicated)

These instructions are under **SVE Integer Binary Arithmetic - 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  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | Zm | Zdn |

### Decode fields

<table>
<thead>
<tr>
<th>R</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>SDIV</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>UDIV</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>SDIVR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>UDIVR</td>
</tr>
</tbody>
</table>

### SVE bitwise logical operations (predicated)

These instructions are under **SVE Integer Binary Arithmetic - 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  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | OPC | Zm | Zdn |

### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ORR (vectors, predicated)</td>
</tr>
<tr>
<td>001</td>
<td>EOR (vectors, predicated)</td>
</tr>
<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 Reduction

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>opcode</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 add reduction (predicated)**

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>opcode</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>SADDV</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>UADDV</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE integer min/max reduction (predicated)**

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>opcode</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>SMAXV</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>UMAXV</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>SMINV</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>UMINV</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE constructive prefix (predicated)**

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>opcode</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>MOVPRFX (predicated)</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE bitwise logical reduction (predicated)

These instructions are under SVE Integer Reduction.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>ORV</td>
</tr>
<tr>
<td>001</td>
<td>FORV</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 - Predicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000100</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op0</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>

SVE bitwise shift by immediate (predicated)

These instructions are under SVE Bitwise Shift - Predicated.

<table>
<thead>
<tr>
<th>opc</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>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>SQSHLU</td>
</tr>
</tbody>
</table>

SVE bitwise shift by vector (predicated)

These instructions are under SVE Bitwise Shift - Predicated.
### SVE bitwise shift by wide elements (predicated)

These instructions are under **SVE Bitwise Shift - Predicated**.

### SVE Integer Unary Arithmetic - Predicated

These instructions are under **SVE encodings**.

### 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>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>CLS</td>
</tr>
<tr>
<td>001</td>
<td>CLZ</td>
</tr>
<tr>
<td>010</td>
<td>CNT</td>
</tr>
<tr>
<td>011</td>
<td>CNOT</td>
</tr>
<tr>
<td>100</td>
<td>FABS</td>
</tr>
<tr>
<td>101</td>
<td>FNEG</td>
</tr>
<tr>
<td>110</td>
<td>NOT (vector)</td>
</tr>
<tr>
<td>111</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE Bitwise Logical - Unpredicated**

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Decode fields op0</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 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>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>

SVE2 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 0</td>
<td>FOR3</td>
</tr>
<tr>
<td>00 1</td>
<td>BSL</td>
</tr>
<tr>
<td>01 0</td>
<td>BCAX</td>
</tr>
<tr>
<td>01 1</td>
<td>BSL1N</td>
</tr>
<tr>
<td>1x 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 1</td>
<td>BSL2N</td>
</tr>
<tr>
<td>11 1</td>
<td>NBSL</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>000</td>
<td>INDEX (immediates)</td>
</tr>
<tr>
<td>010</td>
<td>INDEX (scalar, immediate)</td>
</tr>
<tr>
<td>101</td>
<td>INDEX (immediate, scalar)</td>
</tr>
<tr>
<td>111</td>
<td>INDEX (scalars)</td>
</tr>
</tbody>
</table>

SVE Stack Allocation

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>SVE stack frame adjustment</td>
</tr>
<tr>
<td>011</td>
<td>SVE stack frame size</td>
</tr>
</tbody>
</table>
SVE stack frame adjustment

These instructions are under SVE Stack Allocation.

SVE stack frame size

These instructions are under SVE Stack Allocation.

SVE2 Integer Multiply - Unpredicated

These instructions are under SVE encodings.

SVE2 integer multiply vectors (unpredicated)

These instructions are under SVE2 Integer Multiply - Unpredicated.
### 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>size 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>

### 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>R 0</td>
<td>SODMULH (vectors)</td>
</tr>
<tr>
<td>R 1</td>
<td>SQRDMULH (vectors)</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>op0 0</td>
<td>SVE bitwise shift by wide elements (unpredicated)</td>
</tr>
<tr>
<td>op0 1</td>
<td>SVE bitwise shift by immediate (unpredicated)</td>
</tr>
</tbody>
</table>

### SVE bitwise shift by immediate (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>opc 00</td>
<td>ASR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>opc 01</td>
<td>LSR (wide elements, unpredicated)</td>
</tr>
<tr>
<td>opc 10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>opc 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](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc 00</td>
<td>ASR (immediate, unpredicated)</td>
</tr>
</tbody>
</table>
### SVE address generation

These instructions are under **SVE encodings**.

![Address Generation Table](attachment:address_generation_table.png)

### SVE Integer Misc - Unpredicated

These instructions are under **SVE encodings**.

![Integer Misc Table](attachment:integer_misc_table.png)

### SVE floating-point trig select coefficient

These instructions are under **SVE Integer Misc - Unpredicated**.

![Trig Select Coefficient Table](attachment:trig_select_table.png)

### SVE floating-point exponential accelerator

These instructions are under **SVE Integer Misc - Unpredicated**.

![Exponential Accelerator Table](attachment:exponential_accelerator_table.png)
### SVE constructive prefix (unpredicated)

These instructions are under **SVE Integer Misc - Unpredicated**.

### SVE Element Count

These instructions are under **SVE encodings**.

### SVE saturating inc/dec vector by element count

These instructions are under **SVE Element Count**.
SVE 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>01 0 1</td>
<td>UQINCH (vector)</td>
</tr>
<tr>
<td>01 1 0</td>
<td>SQDECH (vector)</td>
</tr>
<tr>
<td>01 1 1</td>
<td>UQDECH (vector)</td>
</tr>
<tr>
<td>10 0 0</td>
<td>SQINCW (vector)</td>
</tr>
<tr>
<td>10 0 1</td>
<td>UQINCW (vector)</td>
</tr>
<tr>
<td>10 1 0</td>
<td>SQDECW (vector)</td>
</tr>
<tr>
<td>10 1 1</td>
<td>UQDECW (vector)</td>
</tr>
<tr>
<td>11 0 0</td>
<td>SQINCD (vector)</td>
</tr>
<tr>
<td>11 0 1</td>
<td>UQINCD (vector)</td>
</tr>
<tr>
<td>11 1 0</td>
<td>SQDECD (vector)</td>
</tr>
<tr>
<td>11 1 1</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</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>CNTB, CNTD, CNTH, CNTW — CNTB</td>
</tr>
<tr>
<td>01 0</td>
<td>CNTB, CNTD, CNTH, CNTW — CNTH</td>
</tr>
<tr>
<td>10 0</td>
<td>CNTB, CNTD, CNTH, CNTW — CNTH</td>
</tr>
<tr>
<td>11 0</td>
<td>CNTB, CNTD, CNTH, CNTW — CNTH</td>
</tr>
</tbody>
</table>

SVE inc/dec register 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>00 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01 0</td>
<td>INCD, INCH, INCW (vector) — INCH</td>
</tr>
<tr>
<td>01 1</td>
<td>DECD, DECH, DECW (vector) — DECH</td>
</tr>
<tr>
<td>10 0</td>
<td>INCD, INCH, INCW (vector) — INCH</td>
</tr>
<tr>
<td>10 1</td>
<td>DECD, DECH, DECW (vector) — DECH</td>
</tr>
<tr>
<td>11 0</td>
<td>INCD, INCH, INCW (vector) — INCH</td>
</tr>
<tr>
<td>11 1</td>
<td>DECD, DECH, DECW (vector) — DECH</td>
</tr>
</tbody>
</table>

Top-level encodings for A64
### Instruction Details

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0</td>
<td>INCB, INCD, INCH, INCW (scalar) — INCB</td>
</tr>
<tr>
<td>00 1</td>
<td>DECIB, DECD, DECH, DECW (scalar) — DECB</td>
</tr>
<tr>
<td>01 0</td>
<td>INCB, INCD, INCH, INCW (scalar) — INCH</td>
</tr>
<tr>
<td>01 1</td>
<td>DECIB, DECD, DECH, DECW (scalar) — DECH</td>
</tr>
<tr>
<td>10 0</td>
<td>INCB, INCD, INCH, INCW (scalar) — INCW</td>
</tr>
<tr>
<td>10 1</td>
<td>DECIB, DECD, DECH, DECW (scalar) — DECW</td>
</tr>
<tr>
<td>11 0</td>
<td>INCB, INCD, INCH, INCW (scalar) — INCD</td>
</tr>
<tr>
<td>11 1</td>
<td>DECIB, DECD, DECH, DECW (scalar) — DECD</td>
</tr>
</tbody>
</table>

**SVE saturating inc/dec register 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>00 0 0 0 1 0 0</td>
<td>SQINCB — 32-bit</td>
</tr>
<tr>
<td>00 0 0 1 0 1</td>
<td>UQINCB — 32-bit</td>
</tr>
<tr>
<td>00 0 1 0 1 1</td>
<td>SQDECB — 32-bit</td>
</tr>
<tr>
<td>00 0 1 1 0 0</td>
<td>UQDECB — 32-bit</td>
</tr>
<tr>
<td>00 1 0 0 0 1</td>
<td>SQINCB — 64-bit</td>
</tr>
<tr>
<td>00 1 0 0 1 1</td>
<td>UQINCB — 64-bit</td>
</tr>
<tr>
<td>00 1 1 0 1 0</td>
<td>SQDECB — 64-bit</td>
</tr>
<tr>
<td>00 1 1 0 0 1</td>
<td>UQDECB — 64-bit</td>
</tr>
<tr>
<td>01 0 0 0 0 1</td>
<td>SQINCH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 0 0 0 1 1</td>
<td>UQINCH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 0 1 0 0 1</td>
<td>SQDECH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 0 1 0 1 1</td>
<td>UQDECH (scalar) — 32-bit</td>
</tr>
<tr>
<td>01 1 0 0 0 0</td>
<td>SQINCH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 0 0 0 1</td>
<td>UQINCH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 1 0 0 1</td>
<td>SQDECH (scalar) — 64-bit</td>
</tr>
<tr>
<td>01 1 1 0 1 1</td>
<td>UQDECH (scalar) — 64-bit</td>
</tr>
<tr>
<td>10 0 0 0 0 0</td>
<td>SQINW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 0 0 0 0 1</td>
<td>UQINW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 0 1 0 1 1</td>
<td>SQDECW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 0 1 1 0 0</td>
<td>UQDECW (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 1 0 0 0 0</td>
<td>SQINC (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 1 0 0 0 1</td>
<td>UQINC (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 1 1 0 0 1</td>
<td>SQDEC (scalar) — 32-bit</td>
</tr>
<tr>
<td>10 1 1 0 1 1</td>
<td>UQDEC (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 0 0 0 0</td>
<td>SQINCD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 0 0 0 1</td>
<td>UQINCD (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 1 0 0 1</td>
<td>SQDEC (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 0 1 0 1 1</td>
<td>UQDEC (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 1 0 0 0 0</td>
<td>SQIN (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 1 0 0 0 1</td>
<td>UQIN (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 1 1 0 0 1</td>
<td>SQDEC (scalar) — 32-bit</td>
</tr>
<tr>
<td>11 1 1 0 1 1</td>
<td>UQDEC (scalar) — 32-bit</td>
</tr>
</tbody>
</table>
### SVE Bitwise Immediate

These instructions are under [SVE encodings](#).

```
<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>00000101</td>
</tr>
</tbody>
</table>
```

#### Decode fields

- **op0**: DUPL
- **op1**: SVE bitwise logical with immediate (unpredicated)

#### Instruction details

- **! = 11**: UNALLOCATED

The following constraints also apply to this encoding: opc != 11 & opc != 11

#### SVE bitwise logical with immediate (unpredicated)

These instructions are under [SVE Bitwise 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 1 0 1</td>
</tr>
</tbody>
</table>
```

#### Decode fields

- **opc**: ORR (immediate)
- **opc**: EOR (immediate)
- **opc**: AND (immediate)

#### SVE Integer Wide Immediate - Predicated

These instructions are under [SVE encodings](#).

```
<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>00000101</td>
</tr>
</tbody>
</table>
```

#### Decode fields

- **op0**: SVE copy integer immediate (predicated)
- **op0**: UNALLOCATED
- **op0**: FCPY
- **op0**: UNALLOCATED

#### SVE copy integer immediate (predicated)

These instructions are under [SVE Integer Wide Immediate - 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 1 0 1</td>
</tr>
</tbody>
</table>
```
### Decode fields

<table>
<thead>
<tr>
<th>M</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPY (immediate, zeroing)</td>
</tr>
<tr>
<td>1</td>
<td>CPY (immediate, merging)</td>
</tr>
</tbody>
</table>

### SVE Permute Vector - Unpredicated

These instructions are under **SVE encodings**.

![Table](attachment:image.png)

#### Instruction details

- **DUP (scalar)**
- **UNALLOCATED**
- **INSR (scalar)**
- **SVE unpack vector elements**
- **REV (vector)**
- **TBL — SVE**

#### SVE unpack vector elements

These instructions are under **SVE Permute Vector - Unpredicated**.
### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>H</th>
</tr>
</thead>
<tbody>
<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 table lookup (three sources)

These instructions are under **SVE Permute Vector - Unpredicated**.

### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TBL</td>
</tr>
<tr>
<td>1</td>
<td>TBX</td>
</tr>
</tbody>
</table>

### SVE Permute Predicate

These instructions are under **SVE encodings**.

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode fields</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><strong>SVE unpack predicate elements</strong></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>0xxx</td>
<td>xxs0</td>
<td></td>
<td>0</td>
<td><strong>SVE permute predicate elements</strong></td>
</tr>
<tr>
<td>0xxx</td>
<td>xxs1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1010</td>
<td>0000</td>
<td>0</td>
<td></td>
<td>REV (predicate)</td>
</tr>
<tr>
<td>1011</td>
<td>0000</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10x0</td>
<td>1000</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10x0</td>
<td>x100</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10x0</td>
<td>xx10</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10x0</td>
<td>xxx1</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10x1</td>
<td></td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11xx</td>
<td></td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE unpack predicate elements

These instructions are under **SVE Permute Predicate**.

### Decode fields

<table>
<thead>
<tr>
<th>H</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>PUNPKHI, PUNPKLO</strong> — <strong>PUNPKLO</strong></td>
</tr>
</tbody>
</table>
SVE permute predicate elements

These instructions are under **SVE Permute Predicate**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PUNPKHI, PUNPKLO — PUNPKHI</td>
</tr>
</tbody>
</table>

### SVE permute vector elements

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000101</td>
<td>op0</td>
</tr>
</tbody>
</table>

### SVE Permute Vector - Predicated

These instructions are under **SVE encodings**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000101</td>
<td>op0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>1</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>1</td>
<td>0</td>
<td>COMPACT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>0</td>
<td>CPY (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>1</td>
<td></td>
<td></td>
<td>SVE extract element to general register</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>0</td>
<td></td>
<td></td>
<td>SVE extract element to SIMD&amp;FP scalar register</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>0</td>
<td></td>
<td></td>
<td>SVE reverse within elements</td>
</tr>
<tr>
<td>0</td>
<td>01x</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
**SVE extract element to general register**

These instructions are under [SVE Permute Vector - Predicated](#).

![Instruction details](image)

**SVE extract element to SIMD&FP scalar register**

These instructions are under [SVE Permute Vector - Predicated](#).

![Instruction details](image)

**SVE reverse within elements**

These instructions are under [SVE Permute Vector - Predicated](#).

![Instruction details](image)
### SVE conditionally broadcast element to vector

These instructions are under **SVE Permute Vector - Predicated**.

```
0 0 0 0 0 1 0 1 | size 1 0 1 0 0 | B 1 0 0 | Pg Zm | Zdn
```

<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>

### SVE conditionally extract element to SIMD&FP scalar

These instructions are under **SVE Permute Vector - Predicated**.

```
0 0 0 0 0 1 0 1 | size 1 0 1 0 1 | B 1 0 0 | Pg Zm | Vdn
```

<table>
<thead>
<tr>
<th>Decode fields B</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CLASTA (SIMD&amp;FP scalar)</td>
</tr>
<tr>
<td>1</td>
<td>CLASTB (SIMD&amp;FP scalar)</td>
</tr>
</tbody>
</table>

### SVE conditionally extract element to general register

These instructions are under **SVE Permute Vector - Predicated**.

```
0 0 0 0 0 1 0 1 | size 1 1 0 0 0 | B 1 0 1 | Pg Zm | Rdn
```

<table>
<thead>
<tr>
<th>Decode fields 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**.

```
000001010 | op0 1 | 000
```

<table>
<thead>
<tr>
<th>Decode fields 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**.

```
0 0 0 0 0 1 0 1 1 | Zm 0 0 0 | opc2 | Zn | Zd
```

<table>
<thead>
<tr>
<th>Decode fields opc</th>
<th>Zm</th>
<th>opc2</th>
<th>Zn</th>
<th>Zd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>ZIP1, ZIP2 (vectors) — ZIP1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### SVE Integer Compare - Vectors

These instructions are under SVE encodings.

<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>00100100</th>
<th>0</th>
<th>op0</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>opc2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<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>

### Instruction Details

| 0 | SVE integer compare vectors |
| 1 | SVE integer compare with wide elements |

### SVE integer compare vectors

These instructions are under SVE Integer Compare - Vectors.

<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 1 0 0 1 0 0</th>
<th>size 0</th>
<th>Zm</th>
<th>op 0</th>
<th>o2</th>
<th>Pg</th>
<th>Zn</th>
<th>ne</th>
<th>Pd</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>o2</th>
<th>ne</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (vectors) — CMPS</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPEQ</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPEQ</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPS</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (vectors) — CMPS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (vectors) — CMPS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (vectors) — CMPS</td>
</tr>
</tbody>
</table>

### SVE integer compare with wide elements

These instructions are under SVE Integer Compare - Vectors.

<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 1 0 0 1 0 0</th>
<th>size 0</th>
<th>Zm</th>
<th>U</th>
<th>lt</th>
<th>Pg</th>
<th>Zn</th>
<th>ne</th>
<th>Pd</th>
</tr>
</thead>
</table>

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>lt</th>
<th>ne</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPS</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPS</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPS</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPS</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPS</td>
</tr>
</tbody>
</table>
Top-level encodings for A64

### 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>U 1 0 1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPI</td>
</tr>
<tr>
<td>1 1 0</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPO</td>
</tr>
<tr>
<td>1 1 1</td>
<td>CMP&lt;cc&gt; (wide elements) — CMPLS</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>Lt 0 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPS</td>
</tr>
<tr>
<td>0 1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPI</td>
</tr>
<tr>
<td>1 0</td>
<td>CMP&lt;cc&gt; (immediate) — CMPO</td>
</tr>
<tr>
<td>1 1</td>
<td>CMP&lt;cc&gt; (immediate) — CMPLS</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 0 0 0</td>
<td>AND, ANDS (predicates) — not setting the condition flags</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>BIC, BICS (predicates) — not setting the condition flags</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>EOR, EORS (predicates) — not setting the condition flags</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>SEL (predicates)</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>AND, ANDS (predicates) — setting the condition flags</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>BIC, BICS (predicates) — setting the condition flags</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>EOR, EORS (predicates) — setting the condition flags</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### SVE Propagate Break

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
   0100101 00 11 op0
```

<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>1 0 0 0</td>
<td>ORR, ORRS (predicates) — not setting the condition flags</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>ORN, ORNS (predicates) — not setting the condition flags</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>NOR, NORS — not setting the condition flags</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>ORR, ORRS (predicates) — setting the condition flags</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>ORN, ORNS (predicates) — setting the condition flags</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>NOR, NORS — setting the condition flags</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>NAND, NANDS — setting the condition flags</td>
</tr>
</tbody>
</table>

### SVE Propagate break from previous partition

These instructions are under **SVE Propagate Break**.

```
31 30 29 28 27 26 25 24 23 22 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 op S Pm 1 1 Pg 0 Pn B Pd
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op S B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SVE propagate break from previous partition</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE Partition Break

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
   00100101 op0 01 op1 01 op2 op3
```

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>BRKPA, BRKPA — not setting the condition flags</td>
</tr>
<tr>
<td>0 0 1</td>
<td>BRKPB, BRKPBS — not setting the condition flags</td>
</tr>
<tr>
<td>0 1 0</td>
<td>BRKPA, BRKPA — setting the condition flags</td>
</tr>
<tr>
<td>0 1 1</td>
<td>BRKPB, BRKPBS — setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</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 1000</td>
<td>0 0</td>
<td></td>
<td></td>
<td>SVE propagate break to next partition</td>
<td></td>
</tr>
<tr>
<td>0 1000</td>
<td>0 1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 x000</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 x1xx</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 xx1x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 xxx1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>
### SVE propagating break to next partition

These instructions are under **SVE Partition Break**.

| 31 30 29 28 27 26 25 24 23 22 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 | S | 0 1 1 0 0 0 0 1 | Pg | 0 | Pn | 0 | Pdm |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BRKN, BRKNS — not setting the condition flags</td>
</tr>
<tr>
<td>1</td>
<td>BRKN, BRKNS — setting the condition flags</td>
</tr>
</tbody>
</table>

### SVE partition break condition

These instructions are under **SVE Partition Break**.

| 31 30 29 28 27 26 25 24 23 22 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 0 1 | B | S | 0 1 0 0 0 0 0 1 | Pg | 0 | Pn | M | Pd |

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BRKA, BRKAS — not setting the condition flags</td>
</tr>
<tr>
<td>0 1 0</td>
<td>BRKA, BRKAS — setting the condition flags</td>
</tr>
<tr>
<td>1 0</td>
<td>BRKB, BRKBS — not setting the condition flags</td>
</tr>
<tr>
<td>1 1 0</td>
<td>BRKB, BRKBS — setting the condition flags</td>
</tr>
</tbody>
</table>

### SVE Predicate Misc

These instructions are under **SVE encodings**.

<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>00100101</th>
<th>01</th>
<th>op0</th>
<th>11</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
<th>op4</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>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>x0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xx1</td>
<td>x1</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 000 00</td>
<td>0</td>
<td>SVE predicate first active</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 000 != 00</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 100 10 0000</td>
<td>0</td>
<td>SVE predicate zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 100 10 != 0000</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 110 00</td>
<td>0</td>
<td>SVE predicate read from FFR (predicated)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 000 0x</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 000 10</td>
<td>0</td>
<td>PNEXT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 000 11</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
<td></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></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>
<tr>
<td>0</td>
<td>1</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate first active

These instructions are under [SVE Predicate Misc](#).

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>PFIRST</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate zero

These instructions are under [SVE Predicate Misc](#).

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>PFALSE</td>
</tr>
<tr>
<td>0</td>
<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</td>
<td>S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></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</td>
<td>S</td>
</tr>
</tbody>
</table>

### SVE predicate initialize

These instructions are under [SVE Predicate Misc](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### SVE Integer Compare - Scalars

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVE integer compare scalar count and limit</td>
</tr>
<tr>
<td>SVE conditionally terminate scalars</td>
</tr>
<tr>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>SVE pointer conflict compare</td>
</tr>
</tbody>
</table>

### Instruction Details

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>S</td>
</tr>
<tr>
<td>0x</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
</tr>
</tbody>
</table>
SVE integer compare scalar count and limit

These instructions are under **SVE Integer Compare - Scalars**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>lt</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>
<tr>
<td>1</td>
<td>0</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>
</tbody>
</table>

SVE conditionally terminate scalars

These instructions are under **SVE Integer Compare - Scalars**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>ne</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

SVE pointer conflict compare

These instructions are under **SVE Integer Compare - Scalars**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>rw</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

SVE Integer Wide Immediate - Unpredicated

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>op1</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
</tr>
</tbody>
</table>
SVE integer min/max immediate (unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>SVE integer min/max immediate (unpredicated)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>SVE integer multiply immediate (unpredicated)</td>
</tr>
<tr>
<td>1</td>
<td>11 0</td>
<td>SVE broadcast integer immediate (unpredicated)</td>
</tr>
<tr>
<td>1</td>
<td>11 1</td>
<td>SVE broadcast floating-point immediate (unpredicated)</td>
</tr>
</tbody>
</table>

---

SVE integer add/subtract immediate (unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 1 0 1</td>
<td>SVE integer add/subtract immediate (unpredicated)</td>
</tr>
</tbody>
</table>

---

Unallocated

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 0 1 0 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

SVE integer min/max immediate (unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 1 0 0 1</td>
<td>SVE integer min/max immediate (unpredicated)</td>
</tr>
</tbody>
</table>

---

Unallocated

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 0 1 0 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

SVE integer multiply immediate (unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 1 0 0 1</td>
<td>SVE integer multiply immediate (unpredicated)</td>
</tr>
</tbody>
</table>

---

Unallocated

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 0 1 0 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

Unallocated

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 0 1 0 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

Unallocated

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 0 1 1 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

Unallocated

<table>
<thead>
<tr>
<th>Size</th>
<th>Code</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00 0 0 0 0 1 1 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### SVE broadcast integer immediate (unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td><strong>DUP (immediate)</strong></td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE broadcast floating-point immediate (unpredicated)

These instructions are under **SVE Integer Wide Immediate - Unpredicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td><strong>FDUP</strong></td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE predicate count

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><strong>CNTP</strong></td>
</tr>
<tr>
<td>000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### 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>00100101</td>
<td><strong>SVE saturating inc/dec vector by predicate count</strong></td>
</tr>
<tr>
<td>101</td>
<td><strong>SVE saturating inc/dec register by predicate count</strong></td>
</tr>
<tr>
<td>1000</td>
<td><strong>SVE inc/dec vector by predicate count</strong></td>
</tr>
<tr>
<td>1000</td>
<td><strong>SVE inc/dec register by predicate count</strong></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>D  U  opc</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 00</td>
<td>SQINCP (vector)</td>
</tr>
<tr>
<td>0 1 00</td>
<td>UQINCP (vector)</td>
</tr>
<tr>
<td>1 0 00</td>
<td>SQDECP (vector)</td>
</tr>
<tr>
<td>1 1 00</td>
<td>UQDECP (vector)</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  U  sf  op</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>SQINCP (scalar) — 32-bit</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>SQINCP (scalar) — 64-bit</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>UQINCP (scalar) — 32-bit</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>UQINCP (scalar) — 64-bit</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>SQDECP (scalar) — 32-bit</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>SQDECP (scalar) — 64-bit</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>UQDECP (scalar) — 32-bit</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>UQDECP (scalar) — 64-bit</td>
</tr>
</tbody>
</table>

### SVE 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>D  opc2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 00</td>
<td>INCP (vector)</td>
</tr>
<tr>
<td>0 1 00</td>
<td>DECP (vector)</td>
</tr>
<tr>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE 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  opc2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0 00</td>
<td>INCP (vector)</td>
</tr>
<tr>
<td>0 1 00</td>
<td>DECP (vector)</td>
</tr>
</tbody>
</table>
## SVE Write FFR

These instructions are under SVE encodings.

### SVE FFR write from predicate

These instructions are under SVE Write FFR.

### SVE FFR initialise

These instructions are under SVE Write FFR.
SVE Integer Multiply-Add - Unpredicated

These instructions are under SVE encodings.

<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>01000100</td>
</tr>
</tbody>
</table>

**Decode fields**

**Instruction details**

<table>
<thead>
<tr>
<th>op0</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>31 30 29 28 27 26 25 24 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>

**Decode fields**

**Instruction Details**

<table>
<thead>
<tr>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SDOT (vectors)</td>
</tr>
<tr>
<td>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</td>
</tr>
</tbody>
</table>

**Decode fields**

**Instruction Details**

<table>
<thead>
<tr>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SQDMLALBT</td>
</tr>
<tr>
<td>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</td>
</tr>
</tbody>
</table>

**Decode fields**

**Instruction Details**

<table>
<thead>
<tr>
<th>S</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CMLA (vectors)</td>
</tr>
<tr>
<td>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>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>SMLALB (vectors)</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>SMLALT (vectors)</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>UMLALB (vectors)</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>UMLALT (vectors)</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>SMLSLB (vectors)</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>SMLSLT (vectors)</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>UMLSLB (vectors)</td>
</tr>
<tr>
<td>1 1 1 0</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>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>SQDMLALB (vectors)</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>SQDMLALT (vectors)</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>SQDMLSLB (vectors)</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>SQDMLSLT (vectors)</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>0</td>
<td>SQRDMLAH (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>SQRDMLSH (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>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>USDOT (vectors)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</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>op0 10</td>
<td>SVE2 integer pairwise add and accumulate long</td>
</tr>
<tr>
<td>0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>011x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x0x</td>
<td>SVE2 integer unary operations (predicated)</td>
</tr>
<tr>
<td>0xxx</td>
<td>SVE2 saturating/rounding bitwise shift left (predicated)</td>
</tr>
<tr>
<td>10xx 0</td>
<td>SVE2 integer halving add/subtract (predicated)</td>
</tr>
<tr>
<td>10xx 1</td>
<td>SVE2 integer pairwise arithmetic</td>
</tr>
<tr>
<td>11xx 0</td>
<td>SVE2 saturating add/subtract</td>
</tr>
<tr>
<td>11xx 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**SVE2 integer pairwise add and accumulate long**

These instructions are under **SVE2 Integer - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>SADALP</td>
</tr>
<tr>
<td>1</td>
<td>UADALP</td>
</tr>
</tbody>
</table>

**SVE2 integer unary operations (predicated)**

These instructions are under **SVE2 Integer - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x</td>
<td>URECPE</td>
</tr>
<tr>
<td>0 00</td>
<td>URSORTE</td>
</tr>
<tr>
<td>1 00</td>
<td>SQABS</td>
</tr>
<tr>
<td>1 01</td>
<td>SQNEG</td>
</tr>
</tbody>
</table>

**SVE2 saturating/rounding bitwise shift left (predicated)**

These instructions are under **SVE2 Integer - Predicated**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q R N U</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### SVE2 integer halving add/subtract (predicated)

These instructions are under [SVE2 Integer - Predicated](#).

<table>
<thead>
<tr>
<th>Q</th>
<th>R</th>
<th>N</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SRSHL</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>URSHL</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SRSHLR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>URSHLR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SQSHL (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UQSHL (vectors)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SQRSHL</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UQRSRL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SQSHLR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>UQSHLR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SQRSRL</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UQRSRL</td>
</tr>
</tbody>
</table>

### SVE2 integer pairwise arithmetic

These instructions are under [SVE2 Integer - Predicated](#).

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SHADD</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>UHADD</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>SHSUB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UHSUB</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SRHADD</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>URHADD</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>SHSUBR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UHSUBR</td>
</tr>
</tbody>
</table>

### SVE2 saturating add/subtract

These instructions are under [SVE2 Integer - Predicated](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>ADDP</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>SMAXP</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>UMAXP</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>SMINP</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>UMINP</td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Instruction Details

- **SOADD** (vectors, predicated)
- **UQADD** (vectors, predicated)
- **SQSUB** (vectors, predicated)
- **UQSUB** (vectors, predicated)
- **SUQADD**
- **USQADD**
- **SQSUBR**
- **UQSUBR**

### SVE Multiply - Indexed

These instructions are under **SVE encodings**.

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>size</th>
<th>opc</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00001x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>00010x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00011x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>001xxx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100xx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0101xx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0110xx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0111xx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10xxxx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>110xxx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1110xx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11110x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>111111</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Instruction details

- **SVE integer dot product (indexed)**
- **SVE2 integer multiply-add (indexed)**
- **SVE2 saturating multiply-add high (indexed)**
- **SVE mixed sign dot product (indexed)**
- **SVE2 saturating multiply-add (indexed)**
- **SVE2 complex integer dot product (indexed)**
- **SVE2 complex integer multiply-add (indexed)**
- **SVE2 complex saturating multiply-add (indexed)**
- **SVE2 integer multiply-add long (indexed)**
- **SVE2 integer multiply long (indexed)**
- **SVE2 saturating multiply (indexed)**
- **SVE2 saturating multiply high (indexed)**
- **SVE2 integer multiply (indexed)**
- **UNALLOCATED**

### SVE integer dot product (indexed)

These instructions are under **SVE Multiply - Indexed**.

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>SDOT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>UDOT (indexed) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>SDOT (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>UDOT (indexed) — 64-bit</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>0x 0</td>
<td>MLA (indexed) — 16-bit</td>
</tr>
<tr>
<td>0x 1</td>
<td>MLS (indexed) — 16-bit</td>
</tr>
<tr>
<td>10 0</td>
<td>MLA (indexed) — 32-bit</td>
</tr>
<tr>
<td>10 1</td>
<td>MLS (indexed) — 32-bit</td>
</tr>
<tr>
<td>11 0</td>
<td>MLA (indexed) — 64-bit</td>
</tr>
<tr>
<td>11 1</td>
<td>MLS (indexed) — 64-bit</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>0x 0</td>
<td>SQRDMLAH (indexed) — 16-bit</td>
</tr>
<tr>
<td>0x 1</td>
<td>SQRDMLSH (indexed) — 16-bit</td>
</tr>
<tr>
<td>10 0</td>
<td>SQRDMLAH (indexed) — 32-bit</td>
</tr>
<tr>
<td>10 1</td>
<td>SQRDMLSH (indexed) — 32-bit</td>
</tr>
<tr>
<td>11 0</td>
<td>SQRDMLAH (indexed) — 64-bit</td>
</tr>
<tr>
<td>11 1</td>
<td>SQRDMLSH (indexed) — 64-bit</td>
</tr>
</tbody>
</table>

### SVE mixed sign 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>0x 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 0</td>
<td>USDOT (indexed)</td>
</tr>
<tr>
<td>10 1</td>
<td>SUDOT</td>
</tr>
<tr>
<td>11 0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE2 saturating 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>0x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
### Instruction Details

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

#### SVE2 complex 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>Size</td>
<td></td>
</tr>
<tr>
<td>OPC</td>
<td>Zn</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### SVE2 complex 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></td>
</tr>
<tr>
<td>OPC</td>
<td>Zn</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### SVE2 complex saturating 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></td>
</tr>
<tr>
<td>OPC</td>
<td>Zn</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

#### SVE2 integer multiply-add long (indexed)

These instructions are under **SVE Multiply - Indexed**.
### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>S</th>
<th>U</th>
<th>T</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SMLALB (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SMLALT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>UMLALB (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UMLALT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SMLSLB (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>SMLSLT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UMLSLB (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UMLSLT (indexed) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SMLALB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SMLALT (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>UMLALB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UMLALT (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SMLSLB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>SMLSLT (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UMLSLB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UMLSLT (indexed) — 64-bit</td>
</tr>
</tbody>
</table>

### SVE2 integer multiply long (indexed)

These instructions are under **SVE Multiply - Indexed**.

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>U</th>
<th>T</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>SMULLB (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>SMULLT (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>UMULLB (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>UMULLT (indexed) — 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>SMULLB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>SMULLT (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>UMULLB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>UMULLT (indexed) — 64-bit</td>
</tr>
</tbody>
</table>

### SVE2 saturating multiply (indexed)

These instructions are under **SVE Multiply - Indexed**.

### Decode fields

<table>
<thead>
<tr>
<th>size</th>
<th>T</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>SQDMULLB (indexed) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>SQDMULLT (indexed) — 32-bit</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>11 0</td>
<td>SQDMULLB (indexed) — 64-bit</td>
</tr>
<tr>
<td>11 1</td>
<td>SQDMULLT (indexed) — 64-bit</td>
</tr>
</tbody>
</table>

SVE2 integer 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>0x 0</td>
<td>SQDMULH (indexed) — 16-bit</td>
</tr>
<tr>
<td>0x 1</td>
<td>SQRDMULH (indexed) — 16-bit</td>
</tr>
<tr>
<td>10 0</td>
<td>SQDMULH (indexed) — 32-bit</td>
</tr>
<tr>
<td>10 1</td>
<td>SQRDMULH (indexed) — 32-bit</td>
</tr>
<tr>
<td>11 0</td>
<td>SQDMULH (indexed) — 64-bit</td>
</tr>
<tr>
<td>11 1</td>
<td>SQRDMULH (indexed) — 64-bit</td>
</tr>
</tbody>
</table>

SVE2 Widening Integer Arithmetic

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>SVE2 integer add/subtract long</td>
</tr>
<tr>
<td>0x</td>
<td>SVE2 integer add/subtract long</td>
</tr>
<tr>
<td>10</td>
<td>SVE2 integer add/subtract wide</td>
</tr>
<tr>
<td>11</td>
<td>SVE2 integer multiply long</td>
</tr>
</tbody>
</table>

SVE2 integer add/subtract 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>0x</td>
<td>SVE2 integer add/subtract long</td>
</tr>
<tr>
<td>10</td>
<td>SVE2 integer add/subtract wide</td>
</tr>
<tr>
<td>11</td>
<td>SVE2 integer multiply long</td>
</tr>
</tbody>
</table>
### Decode fields & Instruction Details

<table>
<thead>
<tr>
<th>op</th>
<th>S</th>
<th>U</th>
<th>T</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>SADDLB</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>SADDLT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>UADDLB</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UADDLT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SSUBLB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>SSUBLT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>USUBLB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>USUBLT</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SABDLB</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>SABDLT</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>UABDLB</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>UABDLT</td>
</tr>
</tbody>
</table>

### SVE2 integer add/subtract wide

These instructions are under [SVE2 Widening Integer Arithmetic](#).

### SVE2 integer multiply long

These instructions are under [SVE2 Widening Integer Arithmetic](#).
# SVE Misc

These instructions are under [SVE encodings](#).

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>10xx</td>
<td></td>
<td>SVE2 bitwise shift left long</td>
</tr>
<tr>
<td>00xx</td>
<td></td>
<td>SVE2 integer add/subtract interleaved long</td>
</tr>
<tr>
<td>010x</td>
<td></td>
<td>SVE2 bitwise exclusive-or interleaved</td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td>SVE integer matrix multiply accumulate</td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11xx</td>
<td></td>
<td>SVE2 bitwise permute</td>
</tr>
</tbody>
</table>

## SVE2 bitwise shift left long

These instructions are under [SVE Misc](#).

<table>
<thead>
<tr>
<th>U</th>
<th>T</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>SSHLLB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>SSHLLT</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>USHLLB</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>USHLLT</td>
</tr>
</tbody>
</table>

## SVE2 integer add/subtract interleaved long

These instructions are under [SVE Misc](#).

<table>
<thead>
<tr>
<th>S</th>
<th>tb</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>SADDLBT</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>SSUBLBT</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>SSUBLTB</td>
</tr>
</tbody>
</table>

## SVE2 bitwise exclusive-or interleaved

These instructions are under [SVE Misc](#).

<table>
<thead>
<tr>
<th>tb</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FORTBT</td>
</tr>
<tr>
<td>1</td>
<td>FORTTB</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</td>
<td></td>
</tr>
<tr>
<td>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**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>opc</td>
<td></td>
</tr>
<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>

**SVE2 Accumulate**

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>0000</td>
<td>SVE2 complex integer add</td>
</tr>
<tr>
<td>!= 0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00x</td>
<td>SVE2 integer absolute difference and accumulate long</td>
</tr>
<tr>
<td>010</td>
<td>SVE2 integer add/subtract long with carry</td>
</tr>
<tr>
<td>10x</td>
<td>SVE2 bitwise shift right and accumulate</td>
</tr>
<tr>
<td>110</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**.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>CADD</td>
</tr>
<tr>
<td>1</td>
<td>SQCADD</td>
</tr>
</tbody>
</table>
SVE2 integer absolute difference and accumulate long

These instructions are under SVE2 Accumulate.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>U  T</td>
<td></td>
</tr>
<tr>
<td>0  0</td>
<td>SABALB</td>
</tr>
<tr>
<td>0  1</td>
<td>SABALT</td>
</tr>
<tr>
<td>1  0</td>
<td>UABALB</td>
</tr>
<tr>
<td>1  1</td>
<td>UABALT</td>
</tr>
</tbody>
</table>

SVE2 integer add/subtract long with carry

These instructions are under SVE2 Accumulate.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size  T</td>
<td></td>
</tr>
<tr>
<td>0x  0</td>
<td>ADCLB</td>
</tr>
<tr>
<td>0x  1</td>
<td>ADCLT</td>
</tr>
<tr>
<td>1x  0</td>
<td>SBCLB</td>
</tr>
<tr>
<td>1x  1</td>
<td>SBCLT</td>
</tr>
</tbody>
</table>

SVE2 bitwise shift right and accumulate

These instructions are under SVE2 Accumulate.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>R  U</td>
<td></td>
</tr>
<tr>
<td>0  0</td>
<td>SSRA</td>
</tr>
<tr>
<td>0  1</td>
<td>USRA</td>
</tr>
<tr>
<td>1  0</td>
<td>SRSRA</td>
</tr>
<tr>
<td>1  1</td>
<td>URSRA</td>
</tr>
</tbody>
</table>

SVE2 bitwise shift and insert

These instructions are under SVE2 Accumulate.

<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>SRI</td>
</tr>
<tr>
<td>1</td>
<td>SLI</td>
</tr>
</tbody>
</table>

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.

#### SVE2 bitwise shift right narrow

These instructions are under SVE2 Narrowing.
SVE2 integer add/subtract narrow high part

These instructions are under SVE2 Narrowing.

SVE2 character match

These instructions are under SVE encodings.

SVE2 Histogram Computation - Segment

These instructions are under SVE encodings.
SVE2 Crypto Extensions

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>op0</th>
<th>Decode fields</th>
<th>op2</th>
<th>op3</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00 00 00 0000</td>
<td>0000</td>
<td>SVE2 crypto unary operations</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>00 00 != 0000</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>01 0x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>01 11</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x 00</td>
<td>SVE2 crypto destructive binary operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>1x x1</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!=000</td>
<td>0x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!=000</td>
<td>11</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>SVE2 crypto constructive binary operations</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVE2 crypto unary operations

These instructions are under SVE2 Crypto Extensions.

<table>
<thead>
<tr>
<th>size</th>
<th>op</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 1 0 1</td>
<td>op 1 1 0 0</td>
<td>Zdn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>AESMC</td>
</tr>
<tr>
<td>00</td>
<td>AESIMC</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>size</th>
<th>op</th>
<th>o2</th>
<th>Zm</th>
<th>Zdn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 1 0 1</td>
<td>o2</td>
<td>op 1 1 0 0</td>
<td>Zm</td>
<td>Zdn</td>
</tr>
</tbody>
</table>

<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>00</td>
<td>1 0</td>
</tr>
<tr>
<td>00</td>
<td>1 1</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 0</td>
<td>SM4EKEY</td>
</tr>
<tr>
<td>00 1</td>
<td>RAX1</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.

<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>FCVTXNT</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10 00</td>
<td>FCVTNT — single-precision to half-precision</td>
</tr>
<tr>
<td>10 01</td>
<td>FCVTLT — half-precision to single-precision</td>
</tr>
<tr>
<td>10 10</td>
<td>BFVCVTNT</td>
</tr>
<tr>
<td>11 0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11 10</td>
<td>FCVTNT — double-precision to single-precision</td>
</tr>
<tr>
<td>11 11</td>
<td>FCVTLT — single-precision to double-precision</td>
</tr>
</tbody>
</table>

SVE2 floating-point pairwise operations

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>FADDP</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100</td>
<td>FMAXNMP</td>
</tr>
<tr>
<td>101</td>
<td>FMINNMP</td>
</tr>
<tr>
<td>110</td>
<td>FMAXP</td>
</tr>
<tr>
<td>111</td>
<td>FMINP</td>
</tr>
</tbody>
</table>

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](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 0</td>
<td>FMLA (indexed) — half-precision</td>
</tr>
<tr>
<td>0x 1</td>
<td>FMLS (indexed) — half-precision</td>
</tr>
<tr>
<td>10 0</td>
<td>FMLA (indexed) — single-precision</td>
</tr>
<tr>
<td>10 1</td>
<td>FMLS (indexed) — single-precision</td>
</tr>
<tr>
<td>11 0</td>
<td>FMLA (indexed) — double-precision</td>
</tr>
<tr>
<td>11 1</td>
<td>FMLS (indexed) — double-precision</td>
</tr>
</tbody>
</table>

### 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>01100100</td>
<td>SVE BFloat16 floating-point dot product (indexed)</td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</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 encodings](#).
### 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 0 0</td>
<td>FMLALB (indexed)</td>
</tr>
<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 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 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 0</td>
<td>FMLALB (indexed)</td>
</tr>
<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

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>bp0 01100100</td>
</tr>
<tr>
<td>op1</td>
<td>1</td>
</tr>
<tr>
<td>op2</td>
<td>10 bp1 00 bp2</td>
</tr>
</tbody>
</table>
```

### SVE floating-point multiply-add long

```
<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>0 0 0 SVE BFloat16 floating-point dot product</td>
</tr>
<tr>
<td>op1</td>
<td>0 0 1 UNALLOCATED</td>
</tr>
<tr>
<td>op2</td>
<td>0 1 UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>1  SVE floating-point multiply-add long</td>
</tr>
</tbody>
</table>
### SVE floating-point multiply-add long

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>o2 op T</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>FMLALB (vectors)</td>
</tr>
<tr>
<td>0 0 1</td>
<td>FMLALT (vectors)</td>
</tr>
<tr>
<td>0 1 0</td>
<td>FMLSLB (vectors)</td>
</tr>
<tr>
<td>0 1 1</td>
<td>FMLSLT (vectors)</td>
</tr>
<tr>
<td>1 0 0</td>
<td>BFMLALB (vectors)</td>
</tr>
<tr>
<td>1 0 1</td>
<td>BFMLALT (vectors)</td>
</tr>
<tr>
<td>1 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE BF16 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>op</td>
<td></td>
</tr>
<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>o2 op T</td>
<td></td>
</tr>
<tr>
<td>0 0 0</td>
<td>FMLALB (vectors)</td>
</tr>
<tr>
<td>0 0 1</td>
<td>FMLALT (vectors)</td>
</tr>
<tr>
<td>0 1 0</td>
<td>FMLSLB (vectors)</td>
</tr>
<tr>
<td>0 1 1</td>
<td>FMLSLT (vectors)</td>
</tr>
<tr>
<td>1 0 0</td>
<td>BFMLALB (vectors)</td>
</tr>
<tr>
<td>1 0 1</td>
<td>BFMLALT (vectors)</td>
</tr>
<tr>
<td>1 1</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### 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>opc</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
## Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>BFMLA</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>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>FCM&lt;cc&gt; (vectors) — FCMGE</td>
</tr>
<tr>
<td>0</td>
<td>FCM&lt;cc&gt; (vectors) — FCMGT</td>
</tr>
<tr>
<td>0</td>
<td>FCM&lt;cc&gt; (vectors) — FCMEQ</td>
</tr>
<tr>
<td>0</td>
<td>FCM&lt;cc&gt; (vectors) — FCMNE</td>
</tr>
<tr>
<td>0</td>
<td>FCM&lt;cc&gt; (vectors) — FCMUO</td>
</tr>
<tr>
<td>1</td>
<td>FAC&lt;cc&gt; — FACGE</td>
</tr>
<tr>
<td>1</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>opc</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>FRSPRTS</td>
</tr>
</tbody>
</table>

### SVE Floating Point Arithmetic - Predicated

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 arithmetic (predicated)</td>
</tr>
<tr>
<td>10</td>
<td>FTMDAD</td>
</tr>
<tr>
<td>10</td>
<td>! = 000 UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>0000 SVE floating-point arithmetic with immediate (predicated)</td>
</tr>
</tbody>
</table>
### SVE floating-point arithmetic (predicated)

These instructions are under [SVE Floating Point Arithmetic - Predicated](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FADD (vectors, predicated)</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>FMAXNM (vectors)</td>
</tr>
<tr>
<td>0101</td>
<td>FMINNM (vectors)</td>
</tr>
<tr>
<td>0110</td>
<td>FMAX (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>EDIV</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](#).

<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>opc</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01110101</td>
<td></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>opc</th>
<th>Description</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>

<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>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>opc</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>010</td>
<td>FCVTX</td>
</tr>
<tr>
<td>01</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>FCVT — single-precision to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>FCVT — half-precision to single-precision</td>
</tr>
<tr>
<td>10</td>
<td>BFCVT</td>
</tr>
<tr>
<td>11</td>
<td>FCVT — double-precision to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>FCVT — half-precision to double-precision</td>
</tr>
<tr>
<td>10</td>
<td>FCVT — double-precision to single-precision</td>
</tr>
<tr>
<td>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>Description</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 integer convert to floating-point

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 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>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>U</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>0</td>
<td>SCVTF — 16-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
<td>UCVTF — 16-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
<td>SCVTF — 32-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>UCVTF — 32-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>0</td>
<td>SCVTF — 64-bit to half-precision</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
<td>1</td>
<td>UCVTF — 64-bit to half-precision</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>U</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>SCVTF — 32-bit to single-precision</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>UCVTF — 32-bit to single-precision</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>SCVTF — 32-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>UCVTF — 32-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>SCVTF — 64-bit to single-precision</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1</td>
<td>UCVTF — 64-bit to single-precision</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>SCVTF — 64-bit to double-precision</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>UCVTF — 64-bit to double-precision</td>
</tr>
</tbody>
</table>
SVE floating-point recursive reduction

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>opc</th>
<th>opc2</th>
<th>U</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>11</td>
<td>1</td>
<td>FCVTZU — half-precision to 64-bit</td>
</tr>
<tr>
<td>10</td>
<td>0x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>FCVTZS — single-precision to 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>FCVTZU — single-precision to 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>0</td>
<td>FCVTZS — double-precision to 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>00</td>
<td>1</td>
<td>FCVTZU — double-precision to 32-bit</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0</td>
<td>FCVTZS — single-precision to 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>1</td>
<td>FCVTZU — single-precision to 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>FCVTZS — double-precision to 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>1</td>
<td>FCVTZU — double-precision to 64-bit</td>
</tr>
</tbody>
</table>

SVE Floating Point Unary Operations - Unpredicated

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>FADDV</td>
</tr>
<tr>
<td>001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>100</td>
<td>FMAXNMV</td>
</tr>
<tr>
<td>101</td>
<td>FMINNMV</td>
</tr>
<tr>
<td>110</td>
<td>FMAXV</td>
</tr>
<tr>
<td>111</td>
<td>FMINV</td>
</tr>
</tbody>
</table>

SVE floating-point reciprocal estimate (unpredicated)

These instructions are under SVE Floating Point Unary Operations - Unpredicated.

<table>
<thead>
<tr>
<th>opc</th>
<th>Instruction details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SVE floating-point reciprocal estimate (unpredicated)</td>
</tr>
<tr>
<td>!= 00</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>
SVE Floating Point Compare - with Zero

These instructions are under SVE encodings.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>FRECPE</td>
</tr>
<tr>
<td>111</td>
<td>FRSORTE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op0</td>
<td>SVE floating-point compare with zero</td>
</tr>
<tr>
<td>0</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

SVE floating-point compare with zero

These instructions are under SVE Floating Point Compare - with Zero.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>FCM&lt;cc&gt; (zero) — FCMGE</td>
</tr>
<tr>
<td>lt</td>
<td>FCM&lt;cc&gt; (zero) — FCMGT</td>
</tr>
<tr>
<td>ne</td>
<td>FCM&lt;cc&gt; (zero) — FCMLT</td>
</tr>
<tr>
<td></td>
<td>FCM&lt;cc&gt; (zero) — FCMLE</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>FCM&lt;cc&gt; (zero) — FCMEQ</td>
</tr>
<tr>
<td></td>
<td>FCM&lt;cc&gt; (zero) — FCMNE</td>
</tr>
</tbody>
</table>

SVE floating-point serial reduction (predicated)

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>FADDA</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

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>opc</td>
<td>FADDA</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>Decode fields</td>
<td>Instruction details</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
</tr>
<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**.

### SVE floating-point multiply-accumulate writing multiplicand

These instructions are under **SVE Floating Point Multiply-Add**.

### SVE Memory - 32-bit Gather and Unized Contiguous

These instructions are under **SVE encodings**.
| 00 110 0 | SVE contiguous prefetch (scalar plus scalar) |
| 00 111 0 | SVE 32-bit gather prefetch (vector plus immediate) |
| 00 11x 1 | UNALLOCATED |
| 01 1xx   | SVE 32-bit gather load (vector plus immediate) |
| 1x 1xx   | SVE load and broadcast element |

### SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

These instructions are under **SVE Memory - 32-bit Gather and Unsized Contiguous**.

<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 xs 1 Zm 0 msz Pg Rn 0 prfop</td>
</tr>
</tbody>
</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 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>31 30 29 28 27 26 25 24 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 1 xs 1 Zm 0 U ff Pg Rn Zt</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>LD1SH (scalar plus vector)</td>
</tr>
<tr>
<td>0 1</td>
<td>LDFF1SH (scalar plus vector)</td>
</tr>
<tr>
<td>1 0</td>
<td>LD1H (scalar plus vector)</td>
</tr>
<tr>
<td>1 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>31 30 29 28 27 26 25 24 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 xs 1 Zm 0 U ff Pg Rn Zt</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0</td>
<td>LD1W (scalar plus vector)</td>
</tr>
<tr>
<td>1 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>31 30 29 28 27 26 25 24 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 1 imm6 0 msz Pg Rn 0 prfop</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: $\text{opc} \neq 11$ & $\text{opc} \neq 11$

### 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**.
## Decode fields

<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**.

```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 | msz 0 | imm5 1 | 1 | Pg | Zn 0 | prfop
```

## Decode fields

<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**.

```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 | msz 1 | imm5 1 | U | ff | Pg | Zn | Zt
```

## Decode fields

<table>
<thead>
<tr>
<th>msz</th>
<th>U</th>
<th>ff</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>LD1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDFF1SB (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>LD1B (vector plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDFF1B (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>LD1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>LDFF1SH (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>LD1H (vector plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDFF1H (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>LD1W (vector plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDFF1W (vector plus immediate)</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### SVE load and broadcast element

These instructions are under **SVE Memory - 32-bit Gather and Unsized Contiguous**.

```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 1 | dtypeh 1 | imm6 1 | dtype 1 | Pg | Rn | Zt
```

## Decode fields

<table>
<thead>
<tr>
<th>dtypeh</th>
<th>dtype</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00</td>
<td></td>
<td>LD1RB — 8-bit element</td>
</tr>
<tr>
<td>00 01</td>
<td></td>
<td>LD1RB — 16-bit element</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</td>
<td>LD1RB — 32-bit element</td>
</tr>
<tr>
<td>00</td>
<td>LD1RB — 64-bit element</td>
</tr>
<tr>
<td>01</td>
<td>LD1RSW</td>
</tr>
<tr>
<td>01</td>
<td>LD1RH — 16-bit element</td>
</tr>
<tr>
<td>01</td>
<td>LD1RH — 32-bit element</td>
</tr>
<tr>
<td>01</td>
<td>LD1RH — 64-bit element</td>
</tr>
<tr>
<td>10</td>
<td>LD1RSH — 64-bit element</td>
</tr>
<tr>
<td>10</td>
<td>LD1RSH — 32-bit element</td>
</tr>
<tr>
<td>10</td>
<td>LD1RW — 32-bit element</td>
</tr>
<tr>
<td>10</td>
<td>LD1RW — 64-bit element</td>
</tr>
<tr>
<td>11</td>
<td>LD1RSB — 64-bit element</td>
</tr>
<tr>
<td>11</td>
<td>LD1RSB — 32-bit element</td>
</tr>
<tr>
<td>11</td>
<td>LD1RSB — 16-bit element</td>
</tr>
<tr>
<td>11</td>
<td>LD1RD</td>
</tr>
</tbody>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>00</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>!= 00</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>!= 00</td>
<td>1</td>
<td>110</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Instruction details

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>111</td>
<td>SVE contiguous non-temporal load (scalar plus immediate)</td>
</tr>
<tr>
<td>00</td>
<td>110</td>
<td></td>
<td>SVE contiguous non-temporal load (scalar plus scalar)</td>
</tr>
<tr>
<td>!= 00</td>
<td>0</td>
<td>111</td>
<td>SVE load multiple structures (scalar plus immediate)</td>
</tr>
<tr>
<td>!= 00</td>
<td>1</td>
<td>110</td>
<td>SVE load multiple structures (scalar plus scalar)</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td></td>
<td>SVE load and broadcast quadword (scalar plus immediate)</td>
</tr>
<tr>
<td>0</td>
<td>101</td>
<td></td>
<td>SVE contiguous load (scalar plus immediate)</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>101</td>
<td></td>
<td>SVE contiguous non-fault load (scalar plus immediate)</td>
</tr>
<tr>
<td>1</td>
<td>111</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td>SVE load and broadcast quadword (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>SVE contiguous load (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td>SVE contiguous first-fault load (scalar plus scalar)</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>UNALLOCATED</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>msz</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>LDNT1B (scalar plus immediate)</td>
</tr>
<tr>
<td>01</td>
<td>LDNT1H (scalar plus immediate)</td>
</tr>
<tr>
<td>10</td>
<td>LDNT1W (scalar plus immediate)</td>
</tr>
</tbody>
</table>
SVE contiguous non-temporal load (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>msz</td>
<td>LDNT1D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00

SVE load multiple structures (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</td>
<td>LDNT1B (scalar plus scalar)</td>
</tr>
<tr>
<td></td>
<td>LDNT1H (scalar plus scalar)</td>
</tr>
<tr>
<td></td>
<td>LDNT1W (scalar plus scalar)</td>
</tr>
<tr>
<td></td>
<td>LDNT1D (scalar plus scalar)</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00

SVE load multiple structures (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>msz</td>
<td>LD2B (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD3B (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD4B (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD2H (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD3H (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD4H (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD2W (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD3W (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD4W (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD2D (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD3D (scalar plus immediate)</td>
</tr>
<tr>
<td></td>
<td>LD4D (scalar plus immediate)</td>
</tr>
</tbody>
</table>

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](#).

**SVE contiguous load (scalar plus immediate)**

These instructions are under [SVE Memory - Contiguous Load](#).
### SVE contiguous non-fault load (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Load](#).

![Decoder fields](#)

<table>
<thead>
<tr>
<th>dtype</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](#).

![Decoder fields](#)

<table>
<thead>
<tr>
<th>dtype</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>LDNF1D</td>
</tr>
</tbody>
</table>

![Decoder fields](#)

<table>
<thead>
<tr>
<th>msz</th>
<th>ssz</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>LD1ROB (scalar plus scalar)</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>LD1ROB (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>LD1ROQH (scalar plus scalar)</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>LD1ROH (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>LD1ROW (scalar plus scalar)</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>LD1ROW (scalar plus scalar)</td>
</tr>
</tbody>
</table>
### Instruction Details

<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](#).

#### 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>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](#).

#### SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).

#### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

#### SVE 64-bit gather prefetch (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

#### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

#### SVE 64-bit gather prefetch (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

#### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

#### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

#### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).

#### SVE 64-bit gather load (vector plus immediate)

These instructions are under [SVE Memory - 64-bit Gather](#).
### Top-level encodings for A64

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>msz</strong></td>
<td></td>
</tr>
<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**.

```
31 30 29 28 27 26 25 24 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 != 00 | 1 1 | Zm | 1 | U | ff | Pg | Rn | Zt
opc
```

The following constraints also apply to this encoding: opc != 00 && opc != 00

### SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)

These instructions are under **SVE Memory - 64-bit Gather**.

```
31 30 29 28 27 26 25 24 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 != 00 | xs | 1 | Zm | 0 | U | ff | Pg | Rn | Zt
opc
```

The following constraints also apply to this encoding: opc != 00 && opc != 00

### Additional Instruction Details

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>opc</strong></td>
<td></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 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 U ff</td>
<td>LDFF1D (scalar plus vector)</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>PRFB (vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td>PRFH (vector plus immediate)</td>
</tr>
<tr>
<td></td>
<td>PRFW (vector plus immediate)</td>
</tr>
<tr>
<td></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>LDNT1SB</td>
</tr>
<tr>
<td></td>
<td>LDNT1B (vector plus scalar)</td>
</tr>
<tr>
<td></td>
<td>LDNT1H (vector plus scalar)</td>
</tr>
<tr>
<td></td>
<td>LDNT1SW</td>
</tr>
<tr>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>LDNT1D (vector plus scalar)</td>
</tr>
</tbody>
</table>

### Additional Instruction Details

- **LD1SB (vector plus immediate)**
- **LDFF1SB (vector plus immediate)**
- **LD1B (vector plus immediate)**
- **LDFF1B (vector plus immediate)**
- **LD1SH (vector plus immediate)**
- **LDFF1SH (vector plus immediate)**
- **LD1H (vector plus immediate)**
- **LDFF1H (vector plus immediate)**
- **LD1SW (vector plus immediate)**
- **LDFF1SW (vector plus immediate)**
### 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>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](#).

### SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under [SVE Memory - 64-bit Gather](#).
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></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>110</td>
<td>STR (predicate)</td>
</tr>
<tr>
<td>110</td>
<td>STR (vector)</td>
</tr>
<tr>
<td>!110</td>
<td>SVE contiguous store (scalar plus scalar)</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></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>111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>111</td>
<td>ST1D (scalar 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>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>011</td>
<td>SVE2 64-bit scatter non-temporal store (vector plus scalar)</td>
</tr>
</tbody>
</table>
```

The following constraints also apply to this encoding: opc !110 && opc !110
SVE contiguous non-temporal store (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

| 00 | 1 | SVE contiguous non-temporal store (scalar plus scalar) |

SVE2 32-bit scatter non-temporal store (vector plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

| 11 | 0 | UNALLOCATED |

SVE store multiple structures (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

| x1 | 0 | UNALLOCATED |

SVE2 64-bit scatter non-temporal store (vector plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

| 00 | 1 | STNT1B (vector plus scalar) |
| 01 | STNT1H (vector plus scalar) |
| 10 | STNT1W (vector plus scalar) |
| 11 | STNT1D (vector plus scalar) |

Decode fields

<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.

| 00 | 1 | STNT1B (scalar plus scalar) |
| 01 | STNT1H (scalar plus scalar) |
| 10 | STNT1W (scalar plus scalar) |
| 11 | STNT1D (scalar plus scalar) |

Decode fields

<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.

| 00 | 1 | STNT1B (vector plus scalar) |
| 01 | STNT1H (vector plus scalar) |
| 10 | STNT1W (vector plus scalar) |
| 11 | UNALLOCATED |

Decode fields

<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.

| 00 | 1 | STNT1B (scalar plus scalar) |
| 01 | STNT1H (scalar plus scalar) |
| 10 | STNT1W (scalar plus scalar) |
| 11 | UNALLOCATED |

Decode fields

<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>UNALLOCATED</td>
</tr>
</tbody>
</table>

The following constraints also apply to this encoding: opc != 00 && opc != 00
### SVE Memory - Scatter with Optional Sign Extend

These instructions are under **SVE encodings**.

### SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under **SVE Memory - Scatter with Optional Sign Extend**.

### SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

These instructions are under **SVE Memory - Scatter with Optional Sign Extend**.

### SVE 64-bit scatter store (scalar plus packed 32-bit unscaled offsets)

These instructions are under **SVE Memory - Scatter with Optional Sign Extend**.

### SVE 64-bit scatter store (scalar plus packed 32-bit scaled offsets)

These instructions are under **SVE Memory - Scatter with Optional Sign Extend**.
### 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>Decode fields</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>Decode fields</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>UNALLOCATED</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>Decode fields</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 scaled offsets)

These instructions are under SVE Memory - Scatter.

### SVE 64-bit scatter store (vector plus immediate)

These instructions are under SVE Memory - Scatter.

### SVE 32-bit scatter store (vector plus immediate)

These instructions are under SVE Memory - Scatter.

### SVE Memory - Contiguous Store with Immediate Offset

These instructions are under SVE encodings.
### SVE contiguous non-temporal store (scalar plus immediate)

These instructions are under [SVE Memory - Contiguous Store with Immediate Offset](#).

<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](#).

<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](#).

<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>
<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.

### Decode fields

<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>00</td>
<td>00000</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>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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Instruction Details

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>imm12</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></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>o2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ADDG</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>SUBG</td>
<td>ARMv8.5-MemTag</td>
</tr>
</tbody>
</table>

Logical (immediate)

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>N</th>
<th>immr</th>
<th>imms</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>0</td>
<td></td>
<td></td>
<td>AND (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0</td>
<td></td>
<td></td>
<td>ORR (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
<td>EOR (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td>ANDS (immediate) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0</td>
<td></td>
<td></td>
<td>AND (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>0</td>
<td></td>
<td></td>
<td>ORR (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
<td>EOR (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0</td>
<td></td>
<td></td>
<td>ANDS (immediate) — 64-bit</td>
</tr>
</tbody>
</table>

Move wide (immediate)

These instructions are under Data Processing -- Immediate.

<table>
<thead>
<tr>
<th>sf</th>
<th>opc</th>
<th>hw</th>
<th>imm16</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0x</td>
<td>MOVZ</td>
<td>32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0x</td>
<td>MOVK</td>
<td>32-bit</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>0x</td>
<td>MOVN</td>
<td>64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0x</td>
<td>MOVZ</td>
<td>64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>0x</td>
<td>MOVK</td>
<td>64-bit</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>01</td>
<td>0</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>00</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>0xxxxx</td>
<td></td>
<td>Conditional branch (immediate)</td>
</tr>
<tr>
<td>110</td>
<td>0xxxxx</td>
<td></td>
<td>Exception generation</td>
</tr>
<tr>
<td>110</td>
<td>010000011001001111</td>
<td></td>
<td>Hints</td>
</tr>
<tr>
<td>110</td>
<td>0100000110011</td>
<td></td>
<td>Barriers</td>
</tr>
<tr>
<td>110</td>
<td>010000000100</td>
<td></td>
<td>PSTATE</td>
</tr>
<tr>
<td>110</td>
<td>01010000110010</td>
<td></td>
<td>System with result</td>
</tr>
<tr>
<td>110</td>
<td>0100x01xxxxxx</td>
<td></td>
<td>System instructions</td>
</tr>
<tr>
<td>110</td>
<td>0100x1xxxxxx</td>
<td></td>
<td>System register move</td>
</tr>
<tr>
<td>110</td>
<td>1xxxxx</td>
<td></td>
<td>Unconditional branch (register)</td>
</tr>
<tr>
<td>x00</td>
<td></td>
<td></td>
<td>Unconditional branch (register)</td>
</tr>
<tr>
<td>x01</td>
<td>0xxxxx</td>
<td></td>
<td>Compare and branch (immediate)</td>
</tr>
<tr>
<td>x01</td>
<td>1xxxxx</td>
<td></td>
<td>Test and branch (immediate)</td>
</tr>
</tbody>
</table>
### Conditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>o1</th>
<th>imm19</th>
<th>00</th>
<th>cond</th>
</tr>
</thead>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>o1</th>
<th>o0</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>B.cond</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

#### Exception generation

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>opc</th>
<th>imm16</th>
<th>op2</th>
<th>LL</th>
</tr>
</thead>
</table>

#### Decode fields

<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>LL</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1xx</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>01</td>
<td>SVC</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>10</td>
<td>HVC</td>
<td>-</td>
</tr>
<tr>
<td>000</td>
<td>000</td>
<td>11</td>
<td>SMC  Armv8.0-A</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>00</td>
<td>BRK</td>
<td>-</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>00</td>
<td>HLT</td>
<td>-</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>00</td>
<td>TCANCEL</td>
<td>TME</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>01</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>011</td>
<td>000</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>00</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>01</td>
<td>DCPS1</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>10</td>
<td>DCPS2</td>
<td>-</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>11</td>
<td>DCPS3  Armv8.0-A</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>000</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>111</td>
<td>000</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Hints

These instructions are under [Branches, Exception Generating and System instructions](#).

| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | CRm | op2 | 1 | 1 | 1 | 1 | 1 |

Top-level encodings for A64
### Barriers

These instructions are under **Branches, Exception Generating and System instructions**.

#### Top-level encodings for A64

<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>000</td>
<td>HINT</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>001</td>
<td>YIELD</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>010</td>
<td>WFE</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>011</td>
<td>WFI</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>100</td>
<td>SEV</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>101</td>
<td>SEVL</td>
<td>-</td>
</tr>
<tr>
<td>0000</td>
<td>110</td>
<td>DGH</td>
<td>ARMv8.0-DGH</td>
</tr>
<tr>
<td>0000</td>
<td>111</td>
<td>XPACD, XPACI, XPACLRI</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0001</td>
<td>000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA1716</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0001</td>
<td>010</td>
<td>PACIB, PACIB1716, PACIBSP, PACIZB — PACIB176</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0001</td>
<td>100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA1716</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0001</td>
<td>111</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB176</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0010</td>
<td>000</td>
<td>FSB</td>
<td>RAS</td>
</tr>
<tr>
<td>0010</td>
<td>001</td>
<td>PSB CSYNC</td>
<td>SPE</td>
</tr>
<tr>
<td>0010</td>
<td>010</td>
<td>TSB CSYNC</td>
<td>ARMv8.4-Trace</td>
</tr>
<tr>
<td>0010</td>
<td>100</td>
<td>CSDB</td>
<td>-</td>
</tr>
<tr>
<td>0011</td>
<td>000</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIAZ</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>001</td>
<td>PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIASP</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>010</td>
<td>PACIB, PACIB1716, PACIBSP, PACIZB — PACIBZ</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>011</td>
<td>PACIB, PACIB1716, PACIBSP, PACIZB — PACIBSP</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>100</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIAZ</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>101</td>
<td>AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIASP</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>110</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBZ</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0011</td>
<td>111</td>
<td>AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBSP</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0100</td>
<td>xx0</td>
<td>BTI</td>
<td>ARMv8.5-BTI</td>
</tr>
</tbody>
</table>

<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>0000</td>
<td>011</td>
<td>11111</td>
<td>TCOMMIT</td>
<td>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>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>001x</td>
<td>011</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01xx</td>
<td>011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

<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>31</td>
<td>30</td>
<td>29 28</td>
<td>27 26 25 24 23 22</td>
<td>21 20 19 18 17 16 15</td>
</tr>
</tbody>
</table>
## PSTATE

These instructions are under [Branches, Exception Generating and System instructions](#).

<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>0100</td>
<td>100</td>
<td>11111</td>
<td>PSSBB</td>
<td>-</td>
</tr>
<tr>
<td>1xxx</td>
<td>011</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

### System with result

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>! = 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td>MSR (immediate)</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 000</td>
<td>11111</td>
<td>CFINV</td>
<td>ARMv8.4-CondM</td>
<td></td>
</tr>
<tr>
<td>000 001</td>
<td>11111</td>
<td>XAFLAG</td>
<td>ARMv8.5-CondM</td>
<td></td>
</tr>
<tr>
<td>000 010</td>
<td>11111</td>
<td>AXFLAG</td>
<td>ARMv8.5-CondM</td>
<td></td>
</tr>
</tbody>
</table>

### System instructions

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>op1</th>
<th>Decode fields</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>L</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 ! = 0011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0011 ! = 011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0011 ! = 000x 011</td>
<td>TSTART</td>
<td>TME</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 0011 0001 011</td>
<td>TTEST</td>
<td>TME</td>
<td></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>op1</th>
<th>Decode fields</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>! = 00</td>
<td>SYS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>SYSL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Decode fields

<table>
<thead>
<tr>
<th>L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MSR (register)</td>
</tr>
<tr>
<td>1</td>
<td>MRS</td>
</tr>
</tbody>
</table>

### Unconditional branch (register)

These instructions are under [Branches, Exception Generating and System instructions](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>op2</th>
<th>Decode fields</th>
<th>op3</th>
<th>Rn</th>
<th>op4</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>11111</td>
<td>000000</td>
<td>00000</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>11111</td>
<td>000000</td>
<td>00000</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>BR</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>11111</td>
<td>000001</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifier</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0000</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>BRAA, BRAAZ, BRAB, BRABZ — key B, zero modifier</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0000</td>
<td>11111</td>
<td>0000xx</td>
<td>00000</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>BLR</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>11111</td>
<td>0000xx</td>
<td>00000</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>11111</td>
<td>0000xx</td>
<td>1xxxx</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>11111</td>
<td>1xxxxx</td>
<td>00000</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>11111</td>
<td>1xxxxx</td>
<td>00000</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>000001</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>000010</td>
<td>00000</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, zero modifier</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>0000xx</td>
<td>00000</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>0000xx</td>
<td>00000</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>01xxxx</td>
<td>00000</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>11111</td>
<td>1xxxxx</td>
<td>00000</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000000</td>
<td>00000</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>opc</td>
<td>op2</td>
<td>Decode fields</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>0010</td>
<td>11111</td>
<td>000010</td>
<td>! = 11111</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>11111</td>
<td>RETAA, RETAB — RETAA</td>
<td>ARMv8.3-PAuth</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>000011</td>
<td>! = 11111</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>11111</td>
<td>RETAA, RETAB — RETAB</td>
<td>ARMv8.3-PAuth</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>0001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>001xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>01xxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>11111</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000000</td>
<td>! = 11111</td>
<td>! = 00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000000</td>
<td>! = 11111</td>
<td>00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000000</td>
<td>11111</td>
<td>! = 00000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000000</td>
<td>11111</td>
<td>00000</td>
<td>ERET</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000000</td>
<td>ERETAA, ERETAB — ERETAA</td>
<td>ERETAA</td>
<td>ARMv8.3-PAuth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>ERETAA, ERETAB — ERETAB</td>
<td>ARMv8.3-PAuth</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000011</td>
<td>! = 11111</td>
<td>! = 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000011</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>11111</td>
<td>000010</td>
<td>11111</td>
<td>11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>11111</td>
<td>000000</td>
<td>11111</td>
<td>11111</td>
<td>DRPS</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>011x</td>
<td>11111</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11111</td>
<td>000000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
### Unconditional branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

![Unconditional branch (immediate)](table)

#### Compare and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

![Compare and branch (immediate)](table)

#### Test and branch (immediate)

These instructions are under [Branches, Exception Generating and System instructions](#).

![Test and branch (immediate)](table)
### Decode fields

<table>
<thead>
<tr>
<th>op</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TBZ</td>
</tr>
<tr>
<td>1</td>
<td>TBNZ</td>
</tr>
</tbody>
</table>

### Loads and Stores

These instructions are under the **top-level**.

### Advanced SIMD load/store multiple structures

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>opcode</th>
<th>size</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>0001</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>0010</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>0011</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>0100</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>0101</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>0110</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>0111</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>1000</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>1001</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>1010</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>1011</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>1100</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>1101</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>1110</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
<tr>
<td>1111</td>
<td>00001</td>
<td>x0</td>
<td>x1</td>
</tr>
</tbody>
</table>

### Instruction Details

- **Advanced SIMD load/store multiple structures**: 
- **Advanced SIMD load/store multiple structures (post-indexed)**
- **UNALLOCATED**
- **Advanced SIMD load/store single structure**: 
- **Advanced SIMD load/store single structure (post-indexed)**
- **Load/store memory tags**
- **Load/store exclusive**
- **LDAPR/STLR (unscaled immediate)**
- **Load register (literal)**
- **Load/store no-allocate pair (offset)**
- **Load/store register pair (post-indexed)**
- **Load/store register pair (offset)**
- **Load/store register pair (pre-indexed)**
- **Load/store register (unscaled immediate)**
- **Load/store register (immediate post-indexed)**
- **Load/store register (unprivileged)**
- **Load/store register (immediate pre-indexed)**
- **ATOMIC MEMORY OPERATIONS**
- **Load/store register (register offset)**
- **Load/store register (pac)**
- **Load/store register (unsigned immediate)**
<table>
<thead>
<tr>
<th>Decode fields L</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0010</td>
<td>ST1 (multiple structures) — four registers</td>
</tr>
<tr>
<td>0 0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0100</td>
<td>ST3 (multiple structures)</td>
</tr>
<tr>
<td>0 0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 0110</td>
<td>ST1 (multiple structures) — three registers</td>
</tr>
<tr>
<td>0 0111</td>
<td>ST1 (multiple structures) — one register</td>
</tr>
<tr>
<td>0 1000</td>
<td>ST2 (multiple structures)</td>
</tr>
<tr>
<td>0 1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0 1010</td>
<td>ST1 (multiple structures) — two registers</td>
</tr>
<tr>
<td>0 1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0000</td>
<td>LD4 (multiple structures)</td>
</tr>
<tr>
<td>1 0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0010</td>
<td>LD1 (multiple structures) — four registers</td>
</tr>
<tr>
<td>1 0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0100</td>
<td>LD3 (multiple structures)</td>
</tr>
<tr>
<td>1 0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 0110</td>
<td>LD1 (multiple structures) — three registers</td>
</tr>
<tr>
<td>1 0111</td>
<td>LD1 (multiple structures) — one register</td>
</tr>
<tr>
<td>1 1000</td>
<td>LD2 (multiple structures)</td>
</tr>
<tr>
<td>1 1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 1010</td>
<td>LD1 (multiple structures) — two registers</td>
</tr>
<tr>
<td>1 1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1 11xx</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

**Advanced SIMD load/store multiple structures (post-indexed)**

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>L</th>
<th>Decode fields Rm</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>ST4 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>11xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>ST4 (multiple structures) — four registers, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>ST1 (multiple structures) — four registers, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>ST3 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>ST1 (multiple structures) — three registers, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>ST1 (multiple structures) — one register, register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>ST2 (multiple structures) — register offset</td>
</tr>
<tr>
<td>0</td>
<td>!= 11111</td>
<td>ST1 (multiple structures) — two registers, register offset</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>ST4 (multiple structures) — immediate offset</td>
</tr>
</tbody>
</table>
### Instruction Details

<table>
<thead>
<tr>
<th>L</th>
<th>Rm</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11111</td>
<td>0010</td>
<td>ST1 (multiple structures) — four registers, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>0100</td>
<td>ST3 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>0110</td>
<td>ST1 (multiple structures) — three registers, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>0111</td>
<td>ST1 (multiple structures) — one register, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>1000</td>
<td>ST2 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>1010</td>
<td>ST1 (multiple structures) — two registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>!= 11111</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>11111</td>
<td>0000</td>
<td>LD4 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>0010</td>
<td>LD1 (multiple structures) — four registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>0100</td>
<td>LD3 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>0110</td>
<td>LD1 (multiple structures) — three registers, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>0111</td>
<td>LD1 (multiple structures) — one register, immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>1000</td>
<td>LD2 (multiple structures) — immediate offset</td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>1010</td>
<td>LD1 (multiple structures) — two registers, immediate offset</td>
</tr>
</tbody>
</table>

### Advanced SIMD load/store single structure

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Decode fields opcode S size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>000</td>
<td>ST1 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>ST3 (single structure) — 8-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x0 ST1 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>010</td>
<td>x1 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x0 ST3 (single structure) — 16-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
<td>x1 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>00 ST1 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>1x UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
<td>01 ST1 (single structure) — 64-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>00 ST3 (single structure) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>10 UNALLOCATED</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>101</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
<td>1</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</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>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
<td>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
<td>x1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>011</td>
<td>x1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>
### Instruction Details

<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>1</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>x1</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>0</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>LD4R</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD load/store single structure (post-indexed)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>L</th>
<th>R</th>
<th>Rm</th>
<th>opcode</th>
<th>S</th>
<th>size</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>010</td>
<td>x1</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1x</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>10</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>010</td>
<td>x1</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>011</td>
<td>x1</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1x</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>01</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
<td>1</td>
<td>10</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>
</tbody>
</table>

---

Top-level encodings for A64

Page 2842
<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>1</td>
<td></td>
<td>100</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>101</td>
<td>10</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>101</td>
<td>0</td>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>101</td>
<td>1</td>
<td>x1</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>0</td>
<td>00</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>0</td>
<td>01</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>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>0</td>
<td>01</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>0</td>
<td>1</td>
<td></td>
<td>010</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>011</td>
<td>x1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1x</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>01</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></td>
<td>110</td>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>111</td>
<td>1</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>LD1 (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>LD3 (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>LD1 (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>LD3 (single structure) — 16-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>LD1 (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>LD3 (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>LD3 (single structure) — 64-bit, register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>110</td>
<td>0</td>
<td></td>
<td>LD1R — register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>!= 11111</td>
<td>111</td>
<td>0</td>
<td></td>
<td>LD3R — register offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>000</td>
<td></td>
<td></td>
<td>LD1 (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>LD3 (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>LD1 (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>LD3 (single structure) — 16-bit, immediate offset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>100</td>
<td>0</td>
<td>00</td>
<td>LD1 (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>LD3 (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>00</td>
<td>LD3 (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>LD3 (single structure) — 64-bit, immediate offset</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>1</td>
<td>0</td>
<td>111111110</td>
<td>110</td>
<td>0</td>
<td>LD1R — immediate offset</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111111111</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>101</td>
<td>1</td>
<td>x1</td>
<td>UNALLOCATED</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>
</tbody>
</table>

<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>1</td>
<td>1</td>
<td>!= 11111</td>
<td>000</td>
<td>LD2 (single structure) — 8-bit, register offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>!= 11111</td>
<td>000</td>
<td>LD4 (single structure) — 8-bit, register offset</td>
<td></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>0</td>
<td>01</td>
<td>LD2 (single structure) — 64-bit, register offset</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>0</td>
<td>01</td>
<td>LD4 (single structure) — 64-bit, register offset</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>
</tbody>
</table>

### Load/store memory tags

These instructions are under [Loads and Stores](#).

---

<table>
<thead>
<tr>
<th>opc</th>
<th>Decode fields</th>
<th>op2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000000000000</td>
<td>00</td>
<td>STZGM</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>00</td>
<td>000000000000</td>
<td>01</td>
<td>LDG</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>00</td>
<td>000000000000</td>
<td>01</td>
<td>STZG — post-index</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>00</td>
<td>000000000000</td>
<td>01</td>
<td>STZG — signed offset</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>00</td>
<td>000000000000</td>
<td>01</td>
<td>STZG — pre-index</td>
<td>ARMv8.5-MemTag</td>
</tr>
</tbody>
</table>

---

Page 2844
### Load/store exclusive

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>o2</th>
<th>L</th>
<th>o1</th>
<th>o0</th>
<th>Rt2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>!= 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0x</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>!= 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STXR B</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>STLXR B</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASP</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASP</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDX R B</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDAXR B</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASP</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 32-bit CASP</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STLL R B</td>
<td>ARMv8.1-LOR</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>STLR B</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>LDLAR B</td>
<td>ARMv8.1-LOR</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>LDAR B</td>
<td>-</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASAB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASB, CASAB, CASALB, CASLB — CASLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>STXR H</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>STLXR H</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASP</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>size</td>
<td>o2</td>
<td>L</td>
<td>o1</td>
<td>o0</td>
<td>Rt2</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>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASP</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>LDXRH</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>LDAXRH</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>STLLRH</td>
<td>ARMv8.1-LOR</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>STLRH</td>
<td>-</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASH, CASAH, CASALH, CASLH — CASH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASH, CASAH, CASALH, CASLH — CASALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>STXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>STLXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>STXP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>STLXP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>LDXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>LDAXR — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>LDAXP — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>STLLR — 32-bit</td>
<td>ARMv8.1-LOR</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>STLX — 32-bit</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CAS</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>LDAR — 32-bit</td>
<td>ARMv8.1-LOR</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11111</td>
<td>CAS, CASA, CASAL, CASL — 32-bit CASA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</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>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>STXR — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>STLXR — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>STXP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>STLXP — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>LDXR — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>LDAXR — 64-bit</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</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>-</td>
<td>STLLR — 64-bit</td>
<td>ARMv8.1-LOR</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>STLX — 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, CASL — 64-bit CAS</td>
<td>ARMv8.1-LSE</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, CASL — 64-bit CASL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>LDLARH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</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, CASL — 64-bit CASA</td>
<td>ARMv8.1-LSE</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, CASL — 64-bit CASAL</td>
<td>ARMv8.1-LSE</td>
</tr>
</tbody>
</table>
### LDAPR/STLR (unscaled immediate)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>02</th>
<th>Decode fields</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>CAS, CASA, CASAL, CASL — 64-bit CASAL</td>
<td>ARMv8.1-LSE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STLURB</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>00</td>
<td>LDR (literal)</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>00</td>
<td>LDAPURB</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>00</td>
<td>LDAPURSB — 64-bit</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>00</td>
<td>LDAPURSB — 32-bit</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>01</td>
<td>STLRH</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>01</td>
<td>LDRURH</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>01</td>
<td>LDAPURSH — 64-bit</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>01</td>
<td>LDAPURSH — 32-bit</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>10</td>
<td>STLR — 32-bit</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>10</td>
<td>LDRUR — 32-bit</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>10</td>
<td>LDRURSW</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>10</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>STLUR — 64-bit</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>11</td>
<td>LDRUR — 64-bit</td>
<td>ARMv8.4-RCPC</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Load register (literal)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>1</th>
<th>1</th>
<th>V</th>
<th>0</th>
<th>0</th>
<th>imm19</th>
<th>Rt</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LDR (literal) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>LDR (literal, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>LDR (literal) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>LDR (literal, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>10</td>
<td>LDRSW (literal)</td>
</tr>
<tr>
<td>10</td>
<td>LDR (literal, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>11</td>
<td>PRFM (literal)</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Load/store no-allocate pair (offset)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>opc</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>V</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>L</th>
<th>imm7</th>
<th>Rt2</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LDR (literal) — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>LDR (literal, SIMD&amp;FP) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>LDR (literal) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>LDR (literal, SIMD&amp;FP) — 64-bit</td>
</tr>
<tr>
<td>10</td>
<td>LDRSW (literal)</td>
</tr>
<tr>
<td>10</td>
<td>LDR (literal, SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>11</td>
<td>PRFM (literal)</td>
</tr>
<tr>
<td>11</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>STNP — 32-bit</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
<td>LDNP — 32-bit</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>STNP (SIMD&amp;FP) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>LDNP (SIMD&amp;FP) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>STNP (SIMD&amp;FP) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>LDNP (SIMD&amp;FP) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>STNP — 64-bit</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>LDNP — 64-bit</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>STNP (SIMD&amp;FP) — 128-bit</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>LDNP (SIMD&amp;FP) — 128-bit</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</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>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>ARMv8.5-MemTag</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>0</td>
<td>0</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
</tbody>
</table>

---

[Loads and Stores](#): This section provides details on the load and store operations used in ARM architecture, including instructions for handling data movement and memory access in the A64 format.
### Decode fields

<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>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 (pre-indexed)

These instructions are under *Loads and Stores*.

### Decode fields

<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>ARMv8.5-MemTag</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 (unscaled immediate)

These instructions are under *Loads and Stores*.

### 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>STURB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
<td>LDURB</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
<td>LDURSB — 64-bit</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>11</td>
<td>LDURSB — 32-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>STUR (SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>LDUR (SIMD&amp;FP) — 8-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td>STUR (SIMD&amp;FP) — 128-bit</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td>LDUR (SIMD&amp;FP) — 128-bit</td>
</tr>
</tbody>
</table>
Load/store register (immediate 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>size V opc</td>
<td></td>
</tr>
<tr>
<td>01 0 00</td>
<td>STURH</td>
</tr>
<tr>
<td>01 0 01</td>
<td>LDURH</td>
</tr>
<tr>
<td>01 0 10</td>
<td>LDURSH – 64-bit</td>
</tr>
<tr>
<td>01 0 11</td>
<td>LDURSH – 32-bit</td>
</tr>
<tr>
<td>01 1 00</td>
<td>STUR (SIMD&amp;FP) – 16-bit</td>
</tr>
<tr>
<td>01 1 01</td>
<td>LDUR (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>STUR – 32-bit</td>
</tr>
<tr>
<td>10 0 01</td>
<td>LDUR – 32-bit</td>
</tr>
<tr>
<td>10 0 10</td>
<td>LDURSW</td>
</tr>
<tr>
<td>10 1 00</td>
<td>STUR (SIMD&amp;FP) – 32-bit</td>
</tr>
<tr>
<td>10 1 01</td>
<td>LDUR (SIMD&amp;FP) – 32-bit</td>
</tr>
<tr>
<td>11 0 00</td>
<td>STUR – 64-bit</td>
</tr>
<tr>
<td>11 0 01</td>
<td>LDUR – 64-bit</td>
</tr>
<tr>
<td>11 0 10</td>
<td>PREFUM</td>
</tr>
<tr>
<td>11 1 00</td>
<td>STUR (SIMD&amp;FP) – 64-bit</td>
</tr>
<tr>
<td>11 1 01</td>
<td>LDUR (SIMD&amp;FP) – 64-bit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>size V opc</td>
<td></td>
</tr>
<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>
</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>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>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>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<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>LDTRS — 64-bit</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>

---

Top-level encodings for A64
### Atomic memory operations

These instructions are under **Loads and Stores**.

#### Decoded fields

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<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>
<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>1</td>
<td>01</td>
<td>LDR (immediate, SIMD&amp;FP) — 64-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, 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>

---

#### Architecture Version

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>A</th>
<th>R</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>000</td>
<td>LDADDB, LDADDBA, LDADDBALB, LDADDBLB — LDADDB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRBA, LDCLRBA, LDCLRBA, LDCLRBA — LDCLRBA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORB, LDEORB, LDEORB, LDEORB — LDEORB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETB, LDSETB, LDSETB — LDSETB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAXB, LDSMAXB, LDSMAXB, LDSMAXB — LDSMAXB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMINB, LDSMINB, LDSMINB, LDSMINB — LDSMINB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXB, LDUMAXB, LDUMAXB, LDUMAXB — LDUMAXB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMINB, LDUMINB, LDUMINB, LDUMINB — LDUMINB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</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>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADDB, LDADDB, LDADDALB, LDADDLB — LDADDLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLRAB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRRLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEORB, LDEORB, LDEORALB, LDEORLB — LDEORLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADDB, LDADDB, LDADDALB, LDADDLB — LDADDB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRAB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRRLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORB, LDEORB, LDEORALB, LDEORLB — LDEORLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDADDB, LDADDB, LDADDALB, LDADDLB — LDADDB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLRAB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEORB, LDEORB, LDEORALB, LDEORLB — LDEORLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINLB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWPB, SWPAB, SWPALB, SWPLB — SWPB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADDB, LDADDB, LDADDALB, LDADDLB — LDADDB</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</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>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADDDH, LDADDAH, LDADDAALH, LDADDLH — LDADDLH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPLH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADDDH, LDADDAH, LDADDAALH, LDADDLH — LDADDAH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWPH, SWPAH, SWPALH, SWPLH — SWPAH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDADPRH</td>
<td>ARMv8.3-RCPC</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADDDH, LDADDAH, LDADDAALH, LDADDLH — LDADDAH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRLH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORLH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>o3</td>
<td>opc</td>
<td>Instructions Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<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>1</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDBSET, LDBSETAH, LDBSETALH, LDBSETLH — LDBSETALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDBSMAX, LDBSMAXAH, LDBSMAXALH, LDBSMAXLH — LDBSMAXALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDBSMIN, LDBSMINAH, LDBSMINALH, LDBSMINLH — LDBSMINALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDBUMAX, LDBUMAXAH, LDBUMAXALH, LDBUMAXLH — LDBUMAXALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>111</td>
<td>LDBUMIN, LDBUMINAH, LDBUMINALH, LDBUMINLH — LDBUMINALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — SWPALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDAADD, LDAADDA, LDAADDAL, LDADDL — 32-bit LDAADD</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</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>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDAADD, LDAADDA, LDAADDAL, LDADDL — 32-bit LDAADD</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDAADD, LDAADDA, LDAADDAL, LDADDL — 32-bit LDAADD</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWP</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDAADD, LDAADDA, LDAADDAL, LDADDL — 32-bit LDAADD</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDAADD, LDAADDA, LDAADDAL, LDADDL — 32-bit LDAADD</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDBUMAX, LDBUMAXAH, LDBUMAXALH, LDBUMAXLH — LDBUMAXALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>110</td>
<td>LDBUMIN, LDBUMINAH, LDBUMINALH, LDBUMINLH — LDBUMINALH</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWP</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>size</td>
<td>V</td>
<td>A</td>
<td>R</td>
<td>03</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>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100</td>
<td>LDAPR — 32-bit</td>
<td>ARMv8.3-RCPC</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLR, LDCLRL, LDCLRL — 32-bit LDCLRL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 32-bit SWPA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLR, LDCLRL, LDCLRL — 64-bit LDCLRL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>LDCLR, LDCLR, LDCLRL, LDCLRL — 64-bit LDCLRL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWPL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDA</td>
<td>ARMv8.1-LSE</td>
</tr>
</tbody>
</table>
### Load/store register (register offset)

These instructions are under [Loads and Stores](#).

<table>
<thead>
<tr>
<th>size</th>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0 1 0 0 001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 0 010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 0 011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 0 100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 0 101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 0 110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 0 111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 0 1 000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWPA</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 100</td>
<td>LDAPR — 64-bit</td>
<td>ARMv8.3-RCPC</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 000</td>
<td>LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDDAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 001</td>
<td>LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 010</td>
<td>LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 011</td>
<td>LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 100</td>
<td>LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 101</td>
<td>LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINAL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 110</td>
<td>LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 0 111</td>
<td>LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINL</td>
<td>ARMv8.1-LSE</td>
</tr>
<tr>
<td>11</td>
<td>0 1 1 1 000</td>
<td>SWP, SWPA, SWPAL, SWPL — 64-bit SWPA</td>
<td>ARMv8.1-LSE</td>
</tr>
</tbody>
</table>

#### Size

<table>
<thead>
<tr>
<th>size</th>
<th>option</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>!= 011</td>
<td>STRB (register) — extended register</td>
</tr>
<tr>
<td>00</td>
<td>011</td>
<td>STRB (register) — shifted register</td>
</tr>
<tr>
<td>00</td>
<td>!= 011</td>
<td>LDRB (register) — extended register</td>
</tr>
<tr>
<td>00</td>
<td>011</td>
<td>LDRB (register) — shifted register</td>
</tr>
<tr>
<td>00</td>
<td>!= 011</td>
<td>LDRSB (register) — 64-bit with extended register offset</td>
</tr>
<tr>
<td>00</td>
<td>011</td>
<td>LDRSB (register) — 64-bit with shifted register offset</td>
</tr>
<tr>
<td>00</td>
<td>!= 011</td>
<td>LDRSB (register) — 32-bit with extended register offset</td>
</tr>
<tr>
<td>00</td>
<td>011</td>
<td>LDRSB (register) — 32-bit with shifted register offset</td>
</tr>
<tr>
<td>00</td>
<td>!= 011</td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
</tbody>
</table>
### Load/store register (pac)

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>option</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
<td>011</td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>!011</td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>011</td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td></td>
<td>STRH (register)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td></td>
<td>LDRH (register)</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td></td>
<td>LDRSH (register) — 64-bit</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>11</td>
<td></td>
<td>LDRSH (register) — 32-bit</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td></td>
<td>STR (register) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td></td>
<td>LDR (register) — 32-bit</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td></td>
<td>LDRSW (register)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
<td></td>
<td>STR (register) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
<td></td>
<td>LDR (register) — 64-bit</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
<td></td>
<td>PRFM (register)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
<td></td>
<td>STR (register, SIMD&amp;FP)</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td></td>
<td>LDR (register, SIMD&amp;FP)</td>
</tr>
</tbody>
</table>

#### Load/store register (unsigned immediate)

These instructions are under **Loads and Stores**.

<table>
<thead>
<tr>
<th>size</th>
<th>V</th>
<th>opc</th>
<th>imm12</th>
<th>Rn</th>
<th>Rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>size V M W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!= 11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11 0 0 0</td>
<td>LDRAA, LDRAB — key A, offset</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>11 0 0 1</td>
<td>LDRAA, LDRAB — key A, pre-indexed</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>11 0 1 0</td>
<td>LDRAA, LDRAB — key B, offset</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>11 0 1 1</td>
<td>LDRAA, LDRAB — key B, pre-indexed</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>11 1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>Decode fields size</td>
<td>V</td>
<td>opc</td>
</tr>
<tr>
<td>--------------------</td>
<td>---</td>
<td>-----</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>1x</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1x</td>
<td>1</td>
<td>1x</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>

Data Processing -- Register

These instructions are under the top-level.

---

### Decode fields

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 0
- **op1**: 1
- **op2**: 0110
- **op3**: Data-processing (2 source)

---

### Instruction details

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 1
- **op1**: 1
- **op2**: 0110
- **op3**: Data-processing (1 source)

---

### Logical (shifted register)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 0
- **op1**: 0
- **op2**: xxx
- **op3**: Logical (shifted register)

---

### Add/subtract (shifted register)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 0
- **op1**: 1xx0
- **op2**: x
- **op3**: Add/subtract (shifted register)

---

### Add/subtract (extended register)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 0
- **op1**: 1xx1
- **op2**: x
- **op3**: Add/subtract (extended register)

---

### Add/subtract (with carry)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 1
- **op1**: 0000
- **op2**: 000000
- **op3**: Add/subtract (with carry)

---

### Rotate right into flags

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 1
- **op1**: 0000
- **op2**: x00001
- **op3**: Rotate right into flags

---

### Evaluate into flags

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 1
- **op1**: 0000
- **op2**: xx0010
- **op3**: Evaluate into flags

---

### Conditional compare (register)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 1
- **op1**: 0010
- **op2**: xxxx0x
- **op3**: Conditional compare (register)

---

### Conditional compare (immediate)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 1
- **op1**: 0010
- **op2**: xxxx1x
- **op3**: Conditional compare (immediate)

---

### Conditional select

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 1
- **op1**: 0100
- **op2**: x
- **op3**: Conditional select

---

### Data-processing (3 source)

<table>
<thead>
<tr>
<th>op0</th>
<th>op1</th>
<th>op2</th>
<th>op3</th>
</tr>
</thead>
</table>

- **op0**: 1
- **op1**: 1xxx
- **op2**: x
- **op3**: Data-processing (3 source)

---
### Data-processing (2 source)

These instructions are under Data Processing -- Register.

<table>
<thead>
<tr>
<th>sf</th>
<th>0</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>Rm</th>
<th>opcode</th>
<th>Rn</th>
<th>Rd</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>sf</td>
<td>S</td>
<td>opcode</td>
</tr>
<tr>
<td>000001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 00011x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 001101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 00111x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 00001x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 0001xx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 01xxxx</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 000000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0 000010</td>
<td>UDIV — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0 0 000011</td>
<td>SDIV — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0 0 00010x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0 001000</td>
<td>LSLV — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0 0 001001</td>
<td>LSRV — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0 0 001010</td>
<td>ASRV — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0 0 001011</td>
<td>RORV — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0 0 001100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0 010x11</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0 0 010000</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32B</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>0 0 010001</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32H</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>0 0 010010</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32W</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>0 0 010100</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>0 0 010101</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>0 0 010110</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>1 0 000000</td>
<td>SUBP</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>1 0 000010</td>
<td>UDIV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1 0 000011</td>
<td>SDIV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1 0 000100</td>
<td>IRG</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>1 0 000101</td>
<td>GMI</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>1 0 001000</td>
<td>LSLV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1 0 001001</td>
<td>LSRV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1 0 001010</td>
<td>ASRV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1 0 001011</td>
<td>RORV — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1 0 001100</td>
<td>PACGA</td>
<td>ARMv8.3-PAuth</td>
</tr>
<tr>
<td>1 0 010x0x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1 0 010011</td>
<td>CRC32B, CRC32H, CRC32W, CRC32X — CRC32X</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>1 0 010111</td>
<td>CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX</td>
<td>ARMv8.5-MemTag</td>
</tr>
<tr>
<td>1 1 000000</td>
<td>SUBPS</td>
<td>ARMv8.5-MemTag</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>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</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>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</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>1</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>1</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>1</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>1</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</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>1</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>1</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>1</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>1</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>1</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>1</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>1</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>1</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>1</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>1</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>
### Logical (shifted register)

These instructions are under [Data Processing -- 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 opc 0 1 0 1 0 shift N Rm imm6 Rn Rd
```

<table>
<thead>
<tr>
<th>sf</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00 0</td>
<td>AND (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>00 1</td>
<td>BIC (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01 0</td>
<td>ORR (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01 1</td>
<td>ORN (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10 0</td>
<td>EOR (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>10 1</td>
<td>EON (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11 0</td>
<td>ANDS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>11 1</td>
<td>BICS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>00 0</td>
<td>AND (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>00 1</td>
<td>BIC (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01 0</td>
<td>ORR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>01 1</td>
<td>ORN (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10 0</td>
<td>EOR (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>10 1</td>
<td>EON (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11 0</td>
<td>ANDS (shifted register) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>11 1</td>
<td>BICS (shifted register) — 64-bit</td>
</tr>
</tbody>
</table>

### Add/subtract (shifted register)

These instructions are under [Data Processing -- 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 op S 0 1 0 1 1 shift 0 Rm imm6 Rn Rd
```

<table>
<thead>
<tr>
<th>sf</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1xxxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00 0</td>
<td>ADD (shifted register) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>01 0</td>
<td>SUB (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>11 0</td>
<td>ADDS (shifted register) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>11 1</td>
<td>ADD (shifted register) — 64-bit</td>
</tr>
<tr>
<td>sf</td>
<td>op</td>
<td>S</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</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>opt</th>
<th>imm3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>ADD (extended register) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>ADDS (extended register) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>SUB (extended register) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>SUBS (extended register) — 32-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>ADD (extended register) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>00</td>
<td>ADDS (extended register) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>SUB (extended register) — 64-bit</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>SUBS (extended register) — 64-bit</td>
<td></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>opt</th>
<th>imm3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>ADC — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>ADCS — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>SBC — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>SBCS — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>ADC — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>ADCS — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>SBC — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>SBCS — 64-bit</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>opt</th>
<th>imm6</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>00001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>00</td>
<td>00001</td>
</tr>
</tbody>
</table>

---

Top-level encodings for A64
Top-level encodings for A64

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>RMIF</td>
<td>ARMv8.4-CondM</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

**Evaluate into flags**

These instructions are under [Data Processing -- Register](#).

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>opcode2</th>
<th>sz</th>
<th>o3</th>
<th>mask</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></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>SETF8, SETF16 — SETF8</td>
<td>ARMv8.4-CondM</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>SETF8, SETF16 — SETF16</td>
<td>ARMv8.4-CondM</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></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>o2</th>
<th>opcode2</th>
<th>sz</th>
<th>o3</th>
<th>mask</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></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>!=</td>
<td>000000</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>SETF8, SETF16 — SETF8</td>
<td>ARMv8.4-CondM</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00000</td>
<td>0</td>
<td>0</td>
<td>1101</td>
<td>SETF8, SETF16 — SETF16</td>
<td>ARMv8.4-CondM</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

**Conditional compare (immediate)**

These instructions are under [Data Processing -- Register](#).
### Top-level encodings for A64

#### Conditional select

These instructions are under **Data Processing -- Register**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>o3</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) — 32-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) — 32-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>CCMN (immediate) — 64-bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>CCMP (immediate) — 64-bit</td>
</tr>
</tbody>
</table>

#### Data-processing (3 source)

These instructions are under **Data Processing -- Register**.

<table>
<thead>
<tr>
<th>sf</th>
<th>op</th>
<th>S</th>
<th>o2</th>
<th>op2</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>CSEL — 32-bit</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>CSINC — 32-bit</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>CSINV — 32-bit</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>CSNEG — 32-bit</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>CSEL — 64-bit</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>CSINC — 64-bit</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>CSINV — 64-bit</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>CSNEG — 64-bit</td>
</tr>
</tbody>
</table>
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>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0010</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0100</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>Cryptographic AES</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td>Cryptographic three-register SHA</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0101</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>Cryptographic two-register SHA</td>
</tr>
<tr>
<td>0110</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0111</td>
<td>0x</td>
<td>x101</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>00xx</td>
<td>xxx0xxxx1</td>
<td>Advanced SIMD scalar copy</td>
</tr>
<tr>
<td>01x1</td>
<td>01</td>
<td>00xx</td>
<td>xxx0xxxx1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>0111</td>
<td>00xxxxx10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxx0xxxx1</td>
<td>Advanced SIMD scalar three same FP16 ARMv8.2-FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>10xx</td>
<td>xxx01xxx1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>1111</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous ARMv8.2-FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxx0</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx1xxxx1</td>
<td>Advanced SIMD scalar three same extra ARMv8.1-RDMA</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x100</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar two-register miscellaneous</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x110</td>
<td>00xxxxx10</td>
<td>Advanced SIMD scalar pairwise ARMv8.2-FP16</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>1xxxxx10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>x1xxxx10</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxx00</td>
<td>Advanced SIMD scalar three different</td>
</tr>
<tr>
<td>01x1</td>
<td>0x</td>
<td>x1xx</td>
<td>xxxxxxxx1</td>
<td>Advanced SIMD scalar three same</td>
</tr>
<tr>
<td>01x1</td>
<td>10</td>
<td>xxxxxxx1</td>
<td>Advanced SIMD scalar shift by immediate</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>11</td>
<td>xxxxxxx1</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>01x1</td>
<td>1x</td>
<td>xxxxxxx0</td>
<td>Advanced SIMD scalar x indexed element ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>0x</td>
<td>x0xx</td>
<td>xxx0xxxx00</td>
<td>Advanced SIMD table lookup</td>
</tr>
</tbody>
</table>
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](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 00100</td>
<td>AESE</td>
</tr>
<tr>
<td>00 00101</td>
<td>AESD</td>
</tr>
<tr>
<td>00 00110</td>
<td>AESMC</td>
</tr>
<tr>
<td>00 00111</td>
<td>AESIMC</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

---

## Cryptographic two-register SHA

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 00000</td>
<td>SHA1H</td>
</tr>
<tr>
<td>00 00001</td>
<td>SHA1SU1</td>
</tr>
<tr>
<td>00 00010</td>
<td>SHA256SU0</td>
</tr>
<tr>
<td>00 00011</td>
<td>SHA256SU1</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</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>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx1xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>x1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1xxxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>x1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>00 00000</td>
<td>SHA1H</td>
</tr>
<tr>
<td>00 00001</td>
<td>SHA1SU1</td>
</tr>
<tr>
<td>00 00010</td>
<td>SHA256SU0</td>
</tr>
<tr>
<td>00 00011</td>
<td>SHA256SU1</td>
</tr>
<tr>
<td>1x</td>
<td>UNALLOCATED</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>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>imm5</td>
<td>imm4</td>
</tr>
<tr>
<td>0</td>
<td>xxx1</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>xx1x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x1xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>DUP (element)</td>
</tr>
<tr>
<td>0</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>x0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Advanced SIMD scalar two-register miscellaneous FP16

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>op</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>011</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>011</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>111</td>
</tr>
</tbody>
</table>

---

Page 2869
### Advanced SIMD scalar three same extra

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

### Advanced SIMD scalar two-register miscellaneous

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></td>
<td>size</td>
<td>opcode</td>
</tr>
<tr>
<td>0</td>
<td>0000x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0010x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>01111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1000x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>10101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>10111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1100x</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x 011xx</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x 11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x 10110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x 11100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0 00011</td>
<td>SUQADD</td>
</tr>
<tr>
<td>0</td>
<td>0 00111</td>
<td>SQABS</td>
</tr>
<tr>
<td>0</td>
<td>0 01000</td>
<td>CMGT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0 01001</td>
<td>CMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0 01010</td>
<td>CMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>0 01011</td>
<td>ABS</td>
</tr>
<tr>
<td>0</td>
<td>0 10010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0 10100</td>
<td>SQXTN, SQXTN2</td>
</tr>
<tr>
<td>0</td>
<td>0x 10110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x 11010</td>
<td>FCVTNS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x 11011</td>
<td>FCVTMS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x 11100</td>
<td>FCVTAS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0x 11110</td>
<td>SCVTF (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1x 01100</td>
<td>FCMGT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x 01101</td>
<td>FCMEQ (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x 01110</td>
<td>FCMLT (zero)</td>
</tr>
<tr>
<td>0</td>
<td>1x 11010</td>
<td>FCVTPS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1x 11011</td>
<td>FCVTZS (vector, integer)</td>
</tr>
<tr>
<td>0</td>
<td>1x 11101</td>
<td>FRECPE</td>
</tr>
<tr>
<td>0</td>
<td>1x 11111</td>
<td>FRECPX</td>
</tr>
<tr>
<td>1</td>
<td>0 00011</td>
<td>USQADD</td>
</tr>
<tr>
<td>1</td>
<td>0 00111</td>
<td>SQNEG</td>
</tr>
<tr>
<td>1</td>
<td>0 01000</td>
<td>CMGE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>0 01001</td>
<td>CMLE (zero)</td>
</tr>
<tr>
<td>1</td>
<td>0 01010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0 01011</td>
<td>NEG (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1 10010</td>
<td>SQXTUN, SQXTUN2</td>
</tr>
<tr>
<td>1</td>
<td>1 10100</td>
<td>UOXTN, UOXTN2</td>
</tr>
<tr>
<td>1</td>
<td>0x 10110</td>
<td>FCVTXN, FCVTXN2</td>
</tr>
<tr>
<td>1</td>
<td>0x 11010</td>
<td>FCVTNU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x 11011</td>
<td>FCVTMU (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0x 11100</td>
<td>FCVTAU (vector)</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>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>01110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1100x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>111xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>01101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>11011</td>
<td>ADDP (scalar)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01100</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01101</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01111</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01100</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01101</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01111</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01100</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01111</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01100</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01111</td>
</tr>
<tr>
<td>1</td>
<td>11011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01101</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01111</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01111</td>
</tr>
</tbody>
</table>

### Advanced SIMD scalar three different

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).
## Advanced SIMD scalar three same

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

![Instruction Table](#)
<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>0x</td>
<td>11111</td>
<td>FRECPS</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>11111</td>
<td>FRSORTS</td>
</tr>
<tr>
<td>1</td>
<td>00001</td>
<td>UQADD</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00101</td>
<td>USUB</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>USHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01011</td>
<td>UQSHL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>SUB (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10001</td>
<td>CMEQ (register)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10101</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10110</td>
<td>SQRDMLH (vector)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11010</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11100</td>
<td>FCMGE (register)</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11101</td>
<td>FACGE</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>11111</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11010</td>
<td>FABD</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11100</td>
<td>FCMGT (register)</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11101</td>
<td>FACGT</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11110</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>11111</td>
<td>UNALLOCATED</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>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>!= 0000</td>
<td>00001</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>!= 0000</td>
<td>00011</td>
<td>UNALLOCATED</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>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>!= 0000 00101</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 00111</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 01001</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 01011</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 01101</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 01111</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 101xx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 110xx</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 11101</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>!= 0000 11110</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td></td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 00000</td>
<td>00000</td>
<td>SSSH</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 00010</td>
<td>00010</td>
<td>SSRA</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 00100</td>
<td>00100</td>
<td>SRSHR</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 00110</td>
<td>00110</td>
<td>SRSRA</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 01000</td>
<td>01000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 01010</td>
<td>01010</td>
<td>SHL</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 01100</td>
<td>01100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 01110</td>
<td>01110</td>
<td>SOSHL (immediate)</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 10000</td>
<td>10000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 10001</td>
<td>10001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 10010</td>
<td>10010</td>
<td>SQSHRN, SQSHRN2</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 10011</td>
<td>10011</td>
<td>SQSRSHRN, SQSRSHRN2</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 11100</td>
<td>11100</td>
<td>SCVTF (vector, fixed-point)</td>
</tr>
<tr>
<td></td>
<td>0 != 0000 11111</td>
<td>11111</td>
<td>FCVTFS (vector, fixed-point)</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 00000</td>
<td>00000</td>
<td>USHR</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 00010</td>
<td>00010</td>
<td>USRA</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 00100</td>
<td>00100</td>
<td>URSHR</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 00110</td>
<td>00110</td>
<td>URSRA</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 01000</td>
<td>01000</td>
<td>SRI</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 01010</td>
<td>01010</td>
<td>SLI</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 01100</td>
<td>01100</td>
<td>SOSHUL</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 01110</td>
<td>01110</td>
<td>UOSHL (immediate)</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 10000</td>
<td>10000</td>
<td>SQSRSHRUN, SQSRSHRUN2</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 10001</td>
<td>10001</td>
<td>SQSRSHRUN, SQSRSHRUN2</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 10010</td>
<td>10010</td>
<td>SQSRHR, SQSRHR2</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 10011</td>
<td>10011</td>
<td>UQRSHRN, UQRSHRN2</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 11100</td>
<td>11100</td>
<td>UCVT (vector, fixed-point)</td>
</tr>
<tr>
<td></td>
<td>1 != 0000 11111</td>
<td>11111</td>
<td>FCVTZU (vector, fixed-point)</td>
</tr>
</tbody>
</table>

**Advanced SIMD scalar 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>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 0101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>01 1001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 0011</td>
<td>SQDMLAL, SQDMLAL2 (by element)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 0111</td>
<td>SQDMLSL, SQDMLSL2 (by element)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 1011</td>
<td>SORDMULL, SORDMULL2 (by element)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 1100</td>
<td>SORDMULH (by element)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 1101</td>
<td>SORDMULTH (by element)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 1111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 00 0001</td>
<td>FMLA (by element) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0 00 0101</td>
<td>FMLS (by element) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0 1x 0001</td>
<td>FMLA (by element) — single-precision and double-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 1x 0101</td>
<td>FMLS (by element) — single-precision and double-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0 1x 1001</td>
<td>FMUL (by element) — single-precision and double-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 0011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 0111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 1011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 1100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 1101</td>
<td>SQRDMAL (by element)</td>
<td>ARMv8.1-RDMA</td>
<td></td>
</tr>
<tr>
<td>1 1111</td>
<td>SQRDMALS (by element)</td>
<td>ARMv8.1-RDMA</td>
<td></td>
</tr>
<tr>
<td>1 00 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 00 0101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 00 1001</td>
<td>FMULX (by element) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1 1x 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 1x 0101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1 1x 1001</td>
<td>FMULX (by element) — single-precision and double-precision</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Advanced SIMD table lookup

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td>len</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
</tr>
</tbody>
</table>
Advanced SIMD permute

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td>len</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>1x</td>
<td></td>
</tr>
</tbody>
</table>

Advanced SIMD extract

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>op2</td>
<td></td>
</tr>
<tr>
<td>000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>001</td>
<td>UZP1</td>
</tr>
<tr>
<td>010</td>
<td>TRN1</td>
</tr>
<tr>
<td>011</td>
<td>ZIP1</td>
</tr>
<tr>
<td>100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>101</td>
<td>UZP2</td>
</tr>
<tr>
<td>110</td>
<td>TRN2</td>
</tr>
<tr>
<td>111</td>
<td>ZIP2</td>
</tr>
</tbody>
</table>

Advanced SIMD copy

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Q</th>
<th>Decode fields</th>
<th>imm5</th>
<th>imm4</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>op</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0000</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>DUP (element)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>DUP (general)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0010</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD three same (FP16)

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

```
<table>
<thead>
<tr>
<th>Q</th>
<th>Decode fields</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 000</td>
<td>FMAXNM (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0 0 001</td>
<td>FMLA (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0 0 010</td>
<td>FADD (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0 0 011</td>
<td>FMULX</td>
</tr>
<tr>
<td>0</td>
<td>0 0 100</td>
<td>FCMEQ (register)</td>
</tr>
<tr>
<td>0</td>
<td>0 0 101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0 0 110</td>
<td>FMAX (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0 0 111</td>
<td>FRECPS</td>
</tr>
<tr>
<td>0</td>
<td>0 1 000</td>
<td>FMINNM (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0 1 001</td>
<td>FMLS (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0 1 010</td>
<td>FSUB (vector)</td>
</tr>
<tr>
<td>0</td>
<td>0 1 011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0 1 100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0 1 101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>1 0 000</td>
<td>FMIN (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1 0 111</td>
<td>FRSORTS</td>
</tr>
<tr>
<td>1</td>
<td>0 0 000</td>
<td>FMAXNMP (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0 0 001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>0 0 010</td>
<td>FADDP (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0 0 011</td>
<td>FMUL (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0 0 100</td>
<td>FCMGE (register)</td>
</tr>
<tr>
<td>1</td>
<td>0 1 011</td>
<td>FACGE</td>
</tr>
<tr>
<td>1</td>
<td>0 1 100</td>
<td>FMAXP (vector)</td>
</tr>
<tr>
<td>1</td>
<td>0 1 111</td>
<td>FDIV (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1 0 000</td>
<td>FMINNMP (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1 0 001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1 0 010</td>
<td>FABD</td>
</tr>
<tr>
<td>1</td>
<td>1 1 011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1 1 100</td>
<td>FCMGT (register)</td>
</tr>
<tr>
<td>1</td>
<td>1 1 101</td>
<td>FACGT</td>
</tr>
</tbody>
</table>
```
## Advanced SIMD two-register miscellaneous (FP16)

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>00xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>010xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 011xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 11100</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0 0 11000</td>
<td>FRINTN (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 0 11001</td>
<td>FRINTM (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 0 11010</td>
<td>FCVTNS (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 0 11011</td>
<td>FCVTMS (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 0 11100</td>
<td>FCVTAS (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 0 11101</td>
<td>SCVT (vector, integer)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 01100</td>
<td>FCMGT (zero)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 01101</td>
<td>FCMEQ (zero)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 01110</td>
<td>FCMLT (zero)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 01111</td>
<td>FABS (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 11000</td>
<td>FRINTP (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 11001</td>
<td>FRINTZ (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 11010</td>
<td>FCVTPS (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 11011</td>
<td>FCVTZS (vector, integer)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0 1 11101</td>
<td>FRECPE</td>
<td>ARMv8.2-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>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 0 11001</td>
<td>FRINTX (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 0 11010</td>
<td>FCVTNU (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 0 11011</td>
<td>FCVTMU (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 0 11100</td>
<td>FCVTAU (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 0 11101</td>
<td>UCVTF (vector, integer)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 1 01100</td>
<td>FCMGE (zero)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 1 01101</td>
<td>FCMLE (zero)</td>
<td>ARMv8.2-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>ARMv8.2-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>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 1 11010</td>
<td>FCVTPU (vector)</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1 1 11011</td>
<td>FCVTZU (vector, integer)</td>
<td>ARMv8.2-FP16</td>
</tr>
</tbody>
</table>
### Advanced SIMD three-register extension

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>Q</th>
<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>1</td>
<td>0000</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0010</td>
<td>SDOT (vector)</td>
<td>ARMv8.2-DotProd</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0000</td>
<td>SQRDMLAH (vector)</td>
<td>ARMv8.1-RDMA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0001</td>
<td>SQRDMLSH (vector)</td>
<td>ARMv8.1-RDMA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0010</td>
<td>UDOT (vector)</td>
<td>ARMv8.2-DotProd</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10xx</td>
<td>FCMLA</td>
<td>ARMv8.3-CompNum</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11x0</td>
<td>FCADD</td>
<td>ARMv8.3-CompNum</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>BFDO (vector)</td>
<td>ARMv8.2-BF16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
<td>BFMLALB, BFMLALT (vector)</td>
<td>ARMv8.2-BF16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>01xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0xx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>SMMLA (vector)</td>
<td>ARMv8.2-I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>USMMLA (vector)</td>
<td>ARMv8.2-I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>BFMMMLA</td>
<td>ARMv8.2-BF16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>UMMMLA (vector)</td>
<td>ARMv8.2-I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>UNALLOCATED</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.

<table>
<thead>
<tr>
<th>Q</th>
<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>1</td>
<td>0100</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0100</td>
<td>SMMLA (vector)</td>
<td>ARMv8.2-I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1001</td>
<td>USMMLA (vector)</td>
<td>ARMv8.2-I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>01</td>
<td>BFMMMLA</td>
<td>ARMv8.2-BF16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>UMMMLA (vector)</td>
<td>ARMv8.2-I8MM</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10</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>1000x</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x 011XX</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x 10111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x 11110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 10110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00000</td>
<td>REV64</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00001</td>
<td>REV16 (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00010</td>
<td>SADLP</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00011</td>
<td>SUQADD</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00100</td>
<td>CLS (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00101</td>
<td>CNT</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00110</td>
<td>SADALP</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 00111</td>
<td>SQABS</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 01000</td>
<td>CMGT (zero)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 01001</td>
<td>CMEQ (zero)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 01010</td>
<td>CMLT (zero)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 01011</td>
<td>ABS</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10010</td>
<td>XTN, XTN2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10100</td>
<td>SQXTN, SQXTN2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 10110</td>
<td>FCVTN, FCVTN2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 10111</td>
<td>FCVTL, FCVTL2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 11000</td>
<td>PRINTN (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 11001</td>
<td>PRINTM (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 11010</td>
<td>FCVTNS (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 11011</td>
<td>FCVTMS (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 11100</td>
<td>FCVTAS (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 11101</td>
<td>SCVTF (vector, integer)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 11110</td>
<td>PRINT32Z (vector)</td>
<td>ARMv8.5-FPRINT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0x 11111</td>
<td>PRINT64Z (vector)</td>
<td>ARMv8.5-FPRINT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 01100</td>
<td>FCMGT (zero)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 01101</td>
<td>FCEQ (zero)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 01110</td>
<td>FCMLT (zero)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 01111</td>
<td>FABS (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 11000</td>
<td>PRINTP (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 11001</td>
<td>PRINTZ (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 11010</td>
<td>FCVTPS (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 11011</td>
<td>FCVTZS (vector, integer)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 11100</td>
<td>URECPE</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 11101</td>
<td>FRECPE</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1x 11111</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10 10110</td>
<td>BFCVTN, BFCVTN2</td>
<td>ARMv8.2-BF16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00000</td>
<td>REV32 (vector)</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00001</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00010</td>
<td>UADLDP</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00011</td>
<td>USQADD</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Instruction Details

<table>
<thead>
<tr>
<th>Decode fields size opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 00100</td>
<td>CLZ (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 00110</td>
<td>UADALP</td>
<td>-</td>
</tr>
<tr>
<td>1 00111</td>
<td>SQNEG</td>
<td>-</td>
</tr>
<tr>
<td>1 01000</td>
<td>CMGE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1 01001</td>
<td>CMLE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1 01010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 01011</td>
<td>NEG (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 10010</td>
<td>SQXTUN, SQXTUN2</td>
<td>-</td>
</tr>
<tr>
<td>1 10011</td>
<td>Shll, Shll2</td>
<td>-</td>
</tr>
<tr>
<td>1 10100</td>
<td>UQXTN, UQXTN2</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 10110</td>
<td>FCVTXN, FCVTXN2</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 10111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 11000</td>
<td>PRINTA (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 11001</td>
<td>PRINTX (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 11010</td>
<td>FCVTNU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 11011</td>
<td>FCVTMU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 11100</td>
<td>FCVTAU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 11101</td>
<td>UCVTF (vector, integer)</td>
<td>-</td>
</tr>
<tr>
<td>1 0x 11110</td>
<td>PRINT32X (vector)</td>
<td>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>1 0x 11111</td>
<td>PRINT64X (vector)</td>
<td>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>1 00 00101</td>
<td>NOT</td>
<td>-</td>
</tr>
<tr>
<td>1 01 00101</td>
<td>RBIT (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 00101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 01100</td>
<td>FCMGE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 01101</td>
<td>FCMLE (zero)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 01110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 01111</td>
<td>FNEG (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 11000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 11001</td>
<td>PRINTI (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 11010</td>
<td>FCVTPU (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 11011</td>
<td>FCVTZU (vector, integer)</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 11100</td>
<td>URSORTE</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 11101</td>
<td>PRSORTE</td>
<td>-</td>
</tr>
<tr>
<td>1 1x 11111</td>
<td>FSORT (vector)</td>
<td>-</td>
</tr>
<tr>
<td>1 10 10110</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Advanced SIMD across lanes

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

---

### Table for Advanced SIMD across lanes

<table>
<thead>
<tr>
<th>Decode fields size opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>00010</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>001xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0100x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>
Decoded 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>0</td>
<td>00011</td>
<td>SADLV</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</td>
<td>01100</td>
<td>FMAXNMV</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>01111</td>
<td>FMAXV</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>01111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01100</td>
<td>FMINNMV</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>01111</td>
<td>FMINV</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01111</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00011</td>
<td>UADLV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>UMAXV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11010</td>
<td>UMINV</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11011</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01100</td>
<td>FMAXNMV</td>
<td>single-precision and double-precision</td>
</tr>
<tr>
<td>1</td>
<td>0x</td>
<td>01111</td>
<td>FMAXV</td>
<td>single-precision and double-precision</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01100</td>
<td>FMINNMV</td>
<td>single-precision and double-precision</td>
</tr>
<tr>
<td>1</td>
<td>1x</td>
<td>01111</td>
<td>FMINV</td>
<td>single-precision and double-precision</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>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1100</td>
<td>SMULL, SMULL2 (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>SQDMULL, SQDMULL2 (vector)</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>PMULL, PMULL2</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>UADDL, UADDL2</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>UADDW, UADDW2</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>USUBL, USUBL2</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>USUBW, USUBW2</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>RADDHN, RADDHN2</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>UABAL, UABAL2</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>RSUBHN, RSUBHN2</td>
</tr>
<tr>
<td>1</td>
<td>0111</td>
<td>UABDL, UABDL2</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>UMLAL, UMLAL2 (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1010</td>
<td>UMLSL, UMLSL2 (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1011</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1100</td>
<td>UMLULL, UMLULL2 (vector)</td>
</tr>
<tr>
<td>1</td>
<td>1101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

### Advanced SIMD three same

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 size opcode</td>
<td>SHADD</td>
<td>-</td>
</tr>
<tr>
<td>0 00000</td>
<td>SQADD</td>
<td>-</td>
</tr>
<tr>
<td>0 00001</td>
<td>SRHADD</td>
<td>-</td>
</tr>
<tr>
<td>0 00100</td>
<td>SHSUB</td>
<td>-</td>
</tr>
<tr>
<td>0 00101</td>
<td>SQSUB</td>
<td>-</td>
</tr>
<tr>
<td>0 00110</td>
<td>CMGT (register)</td>
<td>-</td>
</tr>
<tr>
<td>0 00111</td>
<td>CMGE (register)</td>
<td>-</td>
</tr>
<tr>
<td>0 01000</td>
<td>SSHL</td>
<td>-</td>
</tr>
<tr>
<td>0 01001</td>
<td>SQSHL (register)</td>
<td>-</td>
</tr>
<tr>
<td>0 01010</td>
<td>SRSHL</td>
<td>-</td>
</tr>
<tr>
<td>0 01011</td>
<td>SQRSHL</td>
<td>-</td>
</tr>
<tr>
<td>0 01100</td>
<td>SMAX</td>
<td>-</td>
</tr>
<tr>
<td>0 01101</td>
<td>SMIN</td>
<td>-</td>
</tr>
<tr>
<td>0 01110</td>
<td>SABD</td>
<td>-</td>
</tr>
<tr>
<td>0 01111</td>
<td>SABA</td>
<td>-</td>
</tr>
<tr>
<td>0 10000</td>
<td>ADD (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 10001</td>
<td>CMTST</td>
<td>-</td>
</tr>
<tr>
<td>0 10010</td>
<td>MLA (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 10011</td>
<td>MUL (vector)</td>
<td>-</td>
</tr>
<tr>
<td>0 10100</td>
<td>SMAXP</td>
<td>-</td>
</tr>
<tr>
<td>0 10101</td>
<td>SMINP</td>
<td>-</td>
</tr>
<tr>
<td>U</td>
<td>Decode fields</td>
<td>opcode</td>
</tr>
<tr>
<td>---</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>10110</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10111</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>11010</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>11100</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>11110</td>
<td></td>
</tr>
<tr>
<td>0 0x</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>0 00</td>
<td>00001</td>
<td></td>
</tr>
<tr>
<td>0 00</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>0 01</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>11010</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>11011</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>11100</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>11110</td>
<td></td>
</tr>
<tr>
<td>0 1x</td>
<td>11111</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>0 11</td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>0 11</td>
<td>11101</td>
<td></td>
</tr>
<tr>
<td>1 00000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 00111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 01111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 10111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Advanced SIMD modified immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

![top-level encodings for A64](image_url)
### Instruction Details

<table>
<thead>
<tr>
<th>Q</th>
<th>Decode fields</th>
<th>op</th>
<th>cmode</th>
<th>o2</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>110x</td>
<td>0</td>
<td></td>
<td>0</td>
<td>MVNI — 32-bit shifting ones</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>0</td>
<td></td>
<td>0</td>
<td>MOVI — 64-bit scalar</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>0</td>
<td></td>
<td>0</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
<td>0</td>
<td></td>
<td>0</td>
<td>MOVI — 64-bit vector</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1111</td>
<td>0</td>
<td></td>
<td>0</td>
<td>FMOV (vector, immediate) — double-precision</td>
<td>-</td>
</tr>
</tbody>
</table>

### Advanced SIMD shift by immediate

These instructions are under [Data Processing -- Scalar Floating-Point and Advanced SIMD](#).

The following constraints also apply to this encoding: immh != 0000 && immh != 0000

### Decode fields

<table>
<thead>
<tr>
<th>U</th>
<th>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110xx</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11110</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00000</td>
<td>SSHR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00010</td>
<td>SSRA</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00100</td>
<td>SRSHR</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>00110</td>
<td>SRSRA</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01000</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01010</td>
<td>SHL</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01100</td>
<td>UNALLOCATED</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>01110</td>
<td>SQSHL (immediate)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10000</td>
<td>SHRN, SHRN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10001</td>
<td>RSHRN, RSHRN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10010</td>
<td>SQSHRN, SQSHRN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10011</td>
<td>SQRSHRN, SQRSHRN2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10100</td>
<td>SSHLL, SSHLL2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11100</td>
<td>SCVTF (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11111</td>
<td>FCVTZS (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000</td>
<td>USHR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00010</td>
<td>USRA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00100</td>
<td>URSHR</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>opcode</td>
<td>Instruction Details</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>---------------------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00110</td>
<td>URSRA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01000</td>
<td>SRI</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01010</td>
<td>SLI</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01100</td>
<td>SOSHLU</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>01110</td>
<td>UQSHL (immediate)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10000</td>
<td>SOSHRUN, SOSHRUN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10001</td>
<td>SORSHRUN, SORSHRUN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10010</td>
<td>UQSHRN, UQSHRN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10011</td>
<td>UQRSHRN, UQRSHRN2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10100</td>
<td>USHLL, USHLL2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11100</td>
<td>UCVTF (vector, fixed-point)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11111</td>
<td>FCVTZU (vector, fixed-point)</td>
<td></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>size</th>
<th>opcode</th>
<th>Instruction Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>0000</td>
<td>01001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0x</td>
<td>0001</td>
<td>0010</td>
<td>SMLAL, SMLAL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>0011</td>
<td>0011</td>
<td>SQDMLAL, SQDMLAL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>0111</td>
<td>SQDMLS, SQDMLSL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>0111</td>
<td>SQDMLSL, SQDMLSL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1010</td>
<td>1100</td>
<td>SQDMLSL (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1101</td>
<td>1110</td>
<td>SDOT (by element)</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>0x</td>
<td>0100</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0000</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0001</td>
<td>FMLA (by element)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>0101</td>
<td>FMLS (by element)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1001</td>
<td>FMUL (by element)</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>1111</td>
<td>SUDOT (by element)</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0001</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>0101</td>
<td>UNALLOCATED</td>
</tr>
<tr>
<td>0</td>
<td>01</td>
<td>1111</td>
<td>BFDOT (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>0001</td>
<td>FMLA (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>0101</td>
<td>FMLS (by element)</td>
</tr>
<tr>
<td>0</td>
<td>1x</td>
<td>1001</td>
<td>FMUL (by element)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0000</td>
<td>FMLAL, FMLAL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0100</td>
<td>FMLSL, FMLSL2 (by element)</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1111</td>
<td>USDOT (by element)</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>0000</td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Architecture Version</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv8.2-DotProd</td>
<td></td>
</tr>
<tr>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>ARMv8.2-I8MM</td>
<td></td>
</tr>
<tr>
<td>ARMv8.2-FHM</td>
<td></td>
</tr>
<tr>
<td>ARMv8.2-FHM</td>
<td></td>
</tr>
<tr>
<td>ARMv8.2-18MM</td>
<td></td>
</tr>
<tr>
<td>ARMv8.2-BF16</td>
<td></td>
</tr>
</tbody>
</table>

---

Top-level encodings for A64
### 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>U size opcode</td>
<td>FMULX (by element) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0x 1100</td>
<td>FCMLA (by element)</td>
<td>ARMv8.3-CompNum</td>
</tr>
<tr>
<td>1 01 0x1</td>
<td>FCMLA (by element)</td>
<td>ARMv8.3-CompNum</td>
</tr>
<tr>
<td>1 1x 1001</td>
<td>FMULX (by element) — single-precision and double-precision</td>
<td>-</td>
</tr>
<tr>
<td>1 10 0x1</td>
<td>FCMLA (by element)</td>
<td>ARMv8.3-CompNum</td>
</tr>
<tr>
<td>1 10 1000</td>
<td>FMLAL, FMLAL2 (by element) — FMLAL2</td>
<td>ARMv8.2-FHM</td>
</tr>
<tr>
<td>1 11 0001</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 11 0011</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 11 0101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 11 0111</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 11 1000</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1 11 1100</td>
<td>UNALLOCATED</td>
<td>-</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>00</td>
<td>SM3TT1A</td>
<td>ARMv8.2-SM3</td>
</tr>
<tr>
<td>01</td>
<td>SM3TT1B</td>
<td>ARMv8.2-SM3</td>
</tr>
<tr>
<td>10</td>
<td>SM3TT2A</td>
<td>ARMv8.2-SM3</td>
</tr>
<tr>
<td>11</td>
<td>SM3TT2B</td>
<td>ARMv8.2-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 Opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SHA512H</td>
<td>ARMv8.2-SHA2</td>
</tr>
<tr>
<td>01</td>
<td>SHA512H2</td>
<td>ARMv8.2-SHA2</td>
</tr>
<tr>
<td>10</td>
<td>SHA512SU1</td>
<td>ARMv8.2-SHA2</td>
</tr>
<tr>
<td>11</td>
<td>RAX1</td>
<td>ARMv8.2-SHA3</td>
</tr>
<tr>
<td>00</td>
<td>SM3PARTW1</td>
<td>ARMv8.2-SM3</td>
</tr>
<tr>
<td>01</td>
<td>SM3PARTW2</td>
<td>ARMv8.2-SM3</td>
</tr>
<tr>
<td>10</td>
<td>SM4EKFY</td>
<td>ARMv8.2-SM4</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Cryptographic two-register SHA 512

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>00</td>
<td>FOR3</td>
<td>ARMv8.2-SHA3</td>
</tr>
<tr>
<td>01</td>
<td>BCAX</td>
<td>ARMv8.2-SHA3</td>
</tr>
<tr>
<td>10</td>
<td>SM3SS1</td>
<td>ARMv8.2-SM3</td>
</tr>
<tr>
<td>11</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Conversion between floating-point and fixed-point

These instructions are under **Data Processing -- Scalar Floating-Point and Advanced SIMD**.

<table>
<thead>
<tr>
<th>sf S ptype rmode</th>
<th>Decode fields Opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x</td>
<td>00x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0x</td>
<td>00x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>00x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>x0</td>
<td>00x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>x1</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0x</td>
<td>00x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1x</td>
<td>01x</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>ptype</td>
<td>rmode</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

**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>Decode fields</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>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td></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>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td></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>
<td></td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></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>11x</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>
<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>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, integer) — 32-bit to single-precision</td>
<td></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>
<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>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>101</td>
<td>FCVTAU (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>110</td>
<td>FMOV (general) — single-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>111</td>
<td>FMOV (general) — 32-bit to single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>01</td>
<td>000</td>
<td>FCVTPS (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>01</td>
<td>001</td>
<td>FCVTPU (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1x</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>000</td>
<td>FCVTMS (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>001</td>
<td>FCVTMU (scalar) — single-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, integer) — single-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, integer) — single-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0x</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, integer) — 32-bit to double-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, integer) — 32-bit to double-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>100</td>
<td>FCVTAS (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>101</td>
<td>FCVTAU (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>000</td>
<td>FCVTPS (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>001</td>
<td>FCVTPU (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>000</td>
<td>FCVTMS (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>001</td>
<td>FCVTMU (scalar) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>000</td>
<td>FCVTZS (scalar, integer) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>001</td>
<td>FCVTZU (scalar, integer) — double-precision to 32-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>110</td>
<td>FCVTZS ARMv8.3-JConv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>111</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>10</td>
<td></td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>000</td>
<td>FCVTNS (scalar) — half-precision to 32-bit ARMv8.2-FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>001</td>
<td>FCVTNU (scalar) — half-precision to 32-bit ARMv8.2-FP16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sf</td>
<td>S</td>
<td>Decode fields</td>
<td>rmode</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>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>010</td>
<td>SCVTF (scalar, integer) — 32-bit to half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>011</td>
<td>UCVTF (scalar, integer) — 32-bit to half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>100</td>
<td>FCTVATAS (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>101</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>110</td>
<td>FMOV (general) — half-precision to 32-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00</td>
<td>111</td>
<td>FMOV (general) — 32-bit to half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01</td>
<td>000</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01</td>
<td>011</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>01</td>
<td>010</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>11</td>
<td>000</td>
<td>FCVT (scalar) — half-precision to 32-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11x</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>000</td>
<td>FCVT (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>001</td>
<td>FCVT (scalar) — single-precision to 64-bit</td>
<td>-</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>
<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>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>FCVT (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>FCVT (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>FCVT (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>FCVT (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>FCVT (scalar) — single-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>x1</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>FCVT (scalar) — double-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>FCVT (scalar) — double-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>FCVT (scalar) — double-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>FCVT (scalar) — double-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>FCVT (scalar) — double-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>1x</td>
<td>UNALLOCATED</td>
<td></td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>FCVT (scalar) — double-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>FCVT (scalar) — double-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>FCVT (scalar) — double-precision to 64-bit</td>
<td>-</td>
<td></td>
<td></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>sf</th>
<th>S</th>
<th>Decode fields</th>
<th>Opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>01 11 001</td>
<td>FCVTZU (scalar, integer) — double-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 x0 11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 01 110</td>
<td>FMOV (general) — top half of 128-bit to 64-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 01 111</td>
<td>FMOV (general) — 64-bit to top half of 128-bit</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>10 1x 11x</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 000</td>
<td>FCVTNS (scalar) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 001</td>
<td>FCVTNU (scalar) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 010</td>
<td>FCVTF (scalar, integer) — 64-bit to half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 011</td>
<td>UCVTF (scalar, integer) — 64-bit to half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 100</td>
<td>FCVTAS (scalar) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 101</td>
<td>FCVTAU (scalar) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 00 111</td>
<td>FMOV (general) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 000</td>
<td>FCVTPS (scalar) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 001</td>
<td>FCVTPU (scalar) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 01 000</td>
<td>FCVTMS (scalar) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 10 001</td>
<td>FCVTMU (scalar) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11 11 001</td>
<td>FCVTZU (scalar, integer) — half-precision to 64-bit</td>
<td>ARMv8.2-FP16</td>
<td></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>S</th>
<th>Decode fields</th>
<th>Opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00 000000</td>
<td>FMOV (register) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 000001</td>
<td>FABS (scalar) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 000010</td>
<td>FNEG (scalar) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 000011</td>
<td>FSORT (scalar) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 000100</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 000101</td>
<td>FCVT — single-precision to double-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 000110</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 000111</td>
<td>FCVT — single-precision to half-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 001000</td>
<td>FRINTN (scalar) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 001001</td>
<td>FRINTP (scalar) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 001010</td>
<td>FRINTM (scalar) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 001011</td>
<td>FRINTZ (scalar) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 001100</td>
<td>FRINTA (scalar) — single-precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00 001101</td>
<td>UNALLOCATED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>S</td>
<td>ptype</td>
<td>opcode</td>
<td>Instruction Details</td>
<td>Architecture Version</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-------</td>
<td>--------</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>001110</td>
<td>FRINTX (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>001111</td>
<td>FRINTI (scalar) — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>010000</td>
<td>FRINT32Z (scalar) — single-precision</td>
<td>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>010001</td>
<td>FRINT32X (scalar) — single-precision</td>
<td>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>010010</td>
<td>FRINT64Z (scalar) — single-precision</td>
<td>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>010011</td>
<td>FRINT64X (scalar) — single-precision</td>
<td>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0101xx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>011xxx</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000000</td>
<td>FMOV (register) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000001</td>
<td>FABS (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000010</td>
<td>FNEG (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000011</td>
<td>FSQRT (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000100</td>
<td>FCVT — double-precision to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000110</td>
<td>BFCVT</td>
<td>ARMv8.2-BF16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>000111</td>
<td>FCVT — double-precision to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010000</td>
<td>FRINTN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010001</td>
<td>FRINTP (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010010</td>
<td>FRINTM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010011</td>
<td>FRINTZ (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010100</td>
<td>FRINTA (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010101</td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>011110</td>
<td>FRINTX (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>011111</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>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010001</td>
<td>FRINT32X (scalar) — double-precision</td>
<td>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010010</td>
<td>FRINT64Z (scalar) — double-precision</td>
<td>ARMv8.5-FRINT</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>010011</td>
<td>FRINT64X (scalar) — double-precision</td>
<td>ARMv8.5-FRINT</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>10</td>
<td>0xxxxx</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>000000</td>
<td>FMOV (register) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>000001</td>
<td>FABS (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>000010</td>
<td>FNEG (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>000011</td>
<td>FSQRT (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>000100</td>
<td>FCVT — half-precision to single-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>000101</td>
<td>FCVT — half-precision to double-precision</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>000110</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>001000</td>
<td>FRINTN (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>001001</td>
<td>FRINTP (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>001010</td>
<td>FRINTM (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>001011</td>
<td>FRINTZ (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>001100</td>
<td>FRINTA (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>001101</td>
<td>UNALLOCATED</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>001110</td>
<td>FRINTX (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>001111</td>
<td>FRINTI (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>01xxxx</td>
<td>UNALLOCATED</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>Decode fields</th>
<th>opcode</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

### Floating-point immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

<table>
<thead>
<tr>
<th>M</th>
<th>S</th>
<th>Decode fields</th>
<th>imm5</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
<td>-</td>
</tr>
</tbody>
</table>

---

Page 2896
### Floating-point conditional 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>op</th>
<th>ndx</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>00000</td>
<td>0</td>
<td>FMOV (scalar, immediate) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></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>op</th>
<th>ndx</th>
<th>Instruction Details</th>
<th>Architecture Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>FCCMP — single-precision</td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>0</td>
<td>FCCMPE — single-precision</td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>FCCMP — double-precision</td>
<td>.</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>FCCMP — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>FCCMPE — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></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>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>001</td>
<td>0101</td>
<td>FMIN (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>0110</td>
<td>FMAXNM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>001</td>
<td>0111</td>
<td>FMINNM (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>1000</td>
<td>FNMUL (scalar) — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0000</td>
<td>FMUL (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0001</td>
<td>FIDIV (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0010</td>
<td>FADD (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0011</td>
<td>FSUB (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0100</td>
<td>FMAX (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0101</td>
<td>FMIN (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0110</td>
<td>FMAXNM (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0111</td>
<td>FMINNM (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1000</td>
<td>FNMUL (scalar) — half-precision</td>
<td>ARMv8.2-FP16</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</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>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>00</td>
<td>00</td>
<td>FMADD — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>01</td>
<td>FMSUB — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>10</td>
<td>FNADD — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>11</td>
<td>FNMSUB — single-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>00</td>
<td>FMADD — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>01</td>
<td>FMSUB — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>FNADD — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>01</td>
<td>11</td>
<td>FNMSUB — double-precision</td>
<td>-</td>
</tr>
<tr>
<td>M</td>
<td>S</td>
<td>ptype</td>
<td>o1</td>
<td>o0</td>
<td>Instruction Details</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-------</td>
<td>----</td>
<td>----</td>
<td>----------------------</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>FMADD — half-precision</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>FMSUB — half-precision</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>FNMADD — half-precision</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>FNMSUB — half-precision</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNALLOCATED</td>
</tr>
</tbody>
</table>

Internal version only: isa v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00

Copyright © 2010-2019 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/VCRMatch/AArch32.VCRMatch

```c
// AArch32.VCRMatch()
// ==================
boolean AArch32.VCRMatch(bits(32) vaddress)

if UsingAArch32() & ELUsingAArch32(EL1) & IsZero(vaddress<1:0>) & 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)
    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);
    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.

if !HaveEL(EL3) & !IsSecure() then return FALSE;
return DBGEN == HIGH & SPIDEN == HIGH;
```

Shared Pseudocode Functions

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress, integer size)
assert ELUsingAArch32(S1TranslationRegime());
assert n <= UInt(DBGDIR.BRPs);

enabled = DBGBCR[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGBCR[n].BT == '0x01';
isbreakpnt = TRUE;
linked_to = FALSE;

state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
linked, DBGBCR[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 = ConstrUnpredictableBool(Unpredictable_BPMATCHHALF);
    if value_mismatch & !mismatch_i then
        value_mismatch = ConstrUnpredictableBool(Unpredictable_BPMISMATCHHALF);

    if vaddress<1> == '1' & DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR[n]+2.
        if value_match then value_match = ConstrUnpredictableBool(Unpredictable_BPMATCHHALF);
        if !value_mismatch then value_mismatch = ConstrUnpredictableBool(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
// 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.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n, bits(32) 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(DBGDIR.BRPs) then
      (c, n) = ConstrainUnpredictableInteger(0, UInt(DBGDIR.BRPs), Unpredictable_BPNOTIMPL);
      assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
      if c == Constraint_DISABLED then return (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 DBGBCR[n].E == '0' then return (FALSE,FALSE);

   context_aware = (n >= UInt(DBGDIR.BRPs) - UInt(DBGDIR.CTX_CMPs));
   // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
   dbgtype = DBGBCR[n].BT;
   if ((dbgtype IN {'011x','11xx'}) && !HaveVirtHostExt()) || // Context matching
      (dbgtype == '010x' && HaltOnBreakpointOrWatchpoint()) || // Address mismatch
      (dbgtype != '0x0x' && !context_aware) ||                         // Context matching
      (dbgtype == '1xxx' && !HaveEL(EL2))) then                        // EL2 extension
      (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
      assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
      if c == Constraint_DISABLED then return (FALSE,FALSE);
   // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

   // Determine what to compare against.
   match_addr = (dbgtype == '0x0x');
   mismatch   = (dbgtype == '010x');
   match_vmid = (dbgtype == '10xx');
   match_cid1 = (dbgtype == '11xx');
   match_cid2 = (dbgtype == '111x');
   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,FALSE);

   // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
   if !linked_to && linked && !match_addr then return (FALSE,FALSE);

   // Do the comparison.
   if match_addr then
      byte = UInt(vaddress<1:0>);
      assert byte IN {0,2};                     // "vaddress" is halfword aligned
      byte_select_match = (DBGBCR[n].BAS<byte> == '1');
      BVR_match = vaddress<31:2> == DBGBVR[n]<31:2> && byte_select_match;
   elsif match_cid1 then
      BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
   elsif match_cid2 then
      BVR_match = (ELUsingAArch32(EL2) && CONTEXTIDR == DBGBVR[n]<31:0>);
   if match_vmid then
      if ELUsingAArch32(EL2) then
         vmid = ZeroExtend(VTBR.VMID, 16);
         bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
      else !Have16bitVMID() || VTCR_EL2.VS == '0' then
         vmid = ZeroExtend(VTBR_EL2.VMID<7:0>, 16);
         bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
      else
         bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
      end
   else
      bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
   end
vmid = VTTBR_EL2.VMID;
bvr_vmid = DBGBoxVR[n]<15:0>;
BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                 vmid == bvr_vmid);
elsif match_cid2 then
  BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                 !ELUsingAArch32(EL2) &&
                 DBGBoxVR[n]<31:0> == CONTEXTIDR_EL2);

bvr_match_valid = (match.addr || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);
return (match && !mismatch, !match && 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
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.
    lbn = UInt(LBN);
    first_ctx_cmp = (UInt(DBGDIDR.BRPs) - UInt(DBGDIDR.CTX_CMPs));
    last_ctx_cmp = UInt(DBGDIDR.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 (Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN);
    case c of
        when Constraint_DISABLED return FALSE;       // Disabled
        when Constraint_NONE linked = FALSE;        // No linking
        // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
if linked then
    vaddress = bits(32) UNKNOWN;
    linked_to = TRUE;
    (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);
return priv_match && security_state_match && (!linked || linked_match);
Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```java
// AArch32.GenerateDebugExceptions()
// ----------------------------------

boolean AArch32.GenerateDebugExceptions()
  return AArch32.GenerateDebugExceptionsFrom(PSTATE_EL, IsSecure());

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

// AArch32.GenerateDebugExceptionsFrom()
// -------------------------------------

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from, boolean secure)
  if from == EL0 && !ELStateUsingAAArch32(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
    spd = if ELUsingAAArch32(EL3) then SDCR.SPDP else MDCR_EL3.SPDP32;
    if spd<1> == '1' then
      enabled = spd<0> == '1';
    else
      // SPD == 0b01 is reserved, but behaves the same as 0b00.
      enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
    if from == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
      enabled = from != EL2;
  return enabled;

Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

// AArch32.CheckForPMUOverflow()
// ---------------------------------
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch32.CheckForPMUOverflow()
  if !ELUsingAAArch32(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 ELUsingAAArch32(EL2) then MDCR_EL2.HPMN else HDRC.HPMN;
      hpme = if ELUsingAAArch32(EL2) then MDCR_EL2.HPME else HDRC.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;
```
// AArch32.CountEvents()
// ===============
// Return TRUE if counter “n” should count its event. For the cycle counter, n == 31.

boolean AArch32.CountEvents(integer n)
assert n == 31 || n < UInt(PMCR.N);

if !ELUsingAArch32(EL1) then return AArch64.CountEvents(n);

// Event counting is disabled in Debug state
debug = Halted();

// In Non-secure state, some counters are reserved for EL2
if HaveEL(EL2) then
    hpmm = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
    hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
    if HaveHPMDExt() then
        hpmd = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMD else HDCR.HPMD;
        E = if n < UInt(hpmm) || n == 31 then PMCR.E else hpme;
    else
        E = PMCR.E;

deploy = E == '1' && PMCNTENSET<n> == '1';

// Event counting in Secure state is prohibited unless any one of:
// * EL3 is not implemented
// * EL3 is using AArch64 and MDCR_EL3.SPME == 1
// * EL3 is using AArch32 and SDCR.SPME == 1
// * Executing at EL0, and SDER.SUNIDEN == 1.
spme = if !ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
prohibited = HaveEL(EL3) && IsSecure() && spme == '0' && (PSTATE.E != EL0 || SDER.SUNIDEN == '0');

// Event counting at EL2 is prohibited if all of:
// * The HPMD Extension is implemented
// * Executing at EL2
// * PMNx is not reserved for EL2
// * HDCR.HPMD == 1
if !prohibited && HaveEL(EL2) && HaveHPMDExt() && PSTATE.E == EL2 && (n < UInt(hpmm) || n == 31) then
    prohibited = (hpmd == '1');

// The IMPLEMENTATION DEFINED authentication interface might override software controls
if prohibited && !HaveNoSecurePMUDisableOverride() then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// For the cycle counter, PMCR.DP enables counting when otherwise prohibited
if prohibited && n == 31 then prohibited = (PMCR.DP == '1');

// Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
filter = if n == 31 then PMCCFILTER 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;
Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

// 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 = HSCTLR.EE;
    PSTATE.II = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSR.ERR = '1';
    UpdateEDSCRFields();
EndOfInstruction();

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

// 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.II = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSR.ERR = '1';
    UpdateEDSCRFields();
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.II = '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();
Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

// AArch32.WatchpointByteMatch()
// -----------------------------------

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
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 // Non contiguous
  byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPASCONTIGUOUS);
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
  end case
  if mask > bottom then
    WVR_match = (vaddress<31:mask> == DBGWVR[n]<31: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_WPMASKEDBITS);
    else
      WVR_match = vaddress<31:bottom> == DBGWVR[n]<31:bottom>;
    end if
  end if
  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, boolean iswrite)
assert ELUsingAArch32(S1TranslationRegime());
assert n <= UInt(DBGDIDR.WRP);

// "ispriv" is FALSE for LDRT/STRT instructions executed at EL1 and all
// load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
// loads.
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 = (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() && HCR2.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.AbortSyndrome

// AArch32.AbortSyndrome()
// ========================
// Creates an exception syndrome record for Abort exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(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(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;
else
   i_syndrome = bits(32) UNKNOWN;
if route_to_monitor then
   IFSR_S = i_syndrome;
else
   IFSR = i_syndrome;
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 DBGDSRext.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);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

// AArch32.TakePrefetchAbortException()
// -------------------------------------

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() && HCR2.TEA == '1' && IsExternalAbort(fault))) ||
(IsDebugException(fault) && HCRD.TDE == '1'));

bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x0C;
lr_offset = 4;

if IsDebugException(fault) then DBGDSRext.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);

Library pseudocode for aarch32/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(EL3) 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);
// AArch32.TakePhysicalFIQException()
// ===========================

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' & !IsHost());
if !route_to_aarch64 & HaveEL(EL3) & !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.FIQ == '1';

if route_to_aarch64 then
    AArch64.TakePhysicalFIQException();
else if 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);

// Take an enabled physical IRQ exception.

AAArch32.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' & !IsHost());
if !route_to_aarch64 & HaveEL(EL3) & !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.IRQ == '1';

if route_to_aarch64 then
    AArch64.TakePhysicalIRQException();
else if 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)

  ClearPendingPhysicalSError();
  // 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' || (!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;

  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, 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);
  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()

  assert 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 IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '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(FALSE, VSESR_EL2.AET, VSESR_EL2.ExT);
    else
        fault = AArch32.AsynchExternalAbort(FALSE, errortype, extflag);

    ClearPendingVirtualSErrorException();
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
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);

AArch32.CheckAdvSIMDOrFPRegisterTraps

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
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccessTrap
    else
      AArch32.SystemAccessTrap(M32_Hyp, 0x8);               // Exception_AdvSIMDFPAccessTrap
    else
      AArch64.AArch32SystemAccessTrap(EL2, 0x8);            // Exception_AdvSIMDFPAccess Trap
// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(int, bit) AArch32.ExceptionClass(Exception exceptype);

il = if ThisInstrLength() == 32 then '1' else '0';

case exceptype of
  when Exception_Uncategorized  ec = 0x00; il = '1';
  when Exception_WFxTrap        ec = 0x01;
  when Exception_CP15RTTrap     ec = 0x03;
  when Exception_CP15RRTTrap    ec = 0x04;
  when Exception_CP14RTTrap     ec = 0x05;
  when Exception_CP14DRTTrap    ec = 0x06;
  when Exception_AdvSIMDFPAccessTrap ec = 0x07;
  when Exception_FPIDTrap      ec = 0x08;
  when Exception_PACTrap       ec = 0x09;
  when Exception_TSTARTAccessTrap ec = 0x1B;
  when Exception_CP14RRTTrap    ec = 0x1C;
  when Exception_BranchTarget  ec = 0x0D;
  when Exception_IllegalState  ec = 0x0E; il = '1';
  when Exception_SupervisorCall ec = 0x11;
  when Exception_HypervisorCall ec = 0x12;
  when Exception_MonitorCall   ec = 0x13;
  when Exception_ERetTrap      ec = 0x1A;
  when Exception_PACFail       ec = 0x1C;
  when Exception_InstructionAbort ec = 0x20; il = '1';
  when Exception_PCAlignment   ec = 0x22; il = '1';
  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;

return (ec,il);

// 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'));
Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception exceptype = exception.exceptype;
    (ec, il) = AArch32.ExceptionClass(exceptype);
    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 exceptype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);
Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

// 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)
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                     // Secure state
    elsif HaveEL(EL2)
        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 and endianness 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)
        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_ae = FALSE;
                element = 0;
                AArch64.FPTrappedException(is_ae, element, 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 = Exception Syndrome(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);
Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCEException

```c
// AArch32.TakeSMCException()
// =========================

AArch32.TakeSMCException()
assert HaveEL(EL3) && ELUsingAArch32(EL3);
AArch32.ITAdvance();
SSAdvance();
bits(32) preferred_exception_return = NextInstrAddr();
vect_offset = 0x08;
lr_offset = 0;
AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

```c
// AArch32.TakeSVCException()
// ==========================

AArch32.TakeSVCException(bits(16) immediate)
AArch32.ITAdvance();
SSAdvance();
route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
bits(32) preferred_exception_return = NextInstrAddr();
vect_offset = 0x08;
lr_offset = 0;
if PSTATE.EL == EL2 || route_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);
```
// 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);
    spsr = GetPSRFromPSTATE();
    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;
    PSTATE.SS = '0';
    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.I = '0';
    PSTATE.I = '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();

// 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;
    spsr = GetPSRFromPSTATE();
    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;
    PSTATE.SS = '0';
    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.I = '0';
    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();
// 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) & ELUsingAArch32(EL3);
  from_secure = IsSecure();
  spsr = GetPSRFromPSTATE();
  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';
  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)
        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;
        else
            cpacr_asedis = CPACR.ASEDIS;
            cpacr_cp10 = CPACR.cp10;
    else
        if HaveEL(EL3) && !IsSecure() then
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';
    endif
    if PSTATE.EL != EL2 then
        // Check if Advanced SIMD disabled in CPACR
        if advsimd && cpacr_asedis == '1' then UNDEFINED;
        if cpacr_cp10 == '10' then
            (c, cpacr_cp10) = ConstrainUnpredictableBits(Unpredictable_RESCPACR);
            // Check if access disabled in CPACR
            if cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '11' disabled = FALSE;
            if disabled then UNDEFINED;
            // If required, check FPEXC enabled bit.
            if fpexc_check && FPEXC.EN == '0' then UNDEFINED;
        endif
    endif
    AArch32.CheckFPAdvSIMDEnabled(); // Also check against HCPTR and CPTR_EL3
AArch64.CheckAdvSIMDOrFPEnabled();
Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

// AArch32.CheckFPAdvSIMDTrap()
// ===================================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if HaveEL(EL2) && !IsSecure() then
            hcptr_tase = HCPtr.TASE;
            hcptr_cp10 = HCPtr.TCP10;
        if HaveEL(EL3) && ELUsingAAArch32(EL3) && !IsSecure() then
            if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
            if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in NSACR
            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';
                exception.syndrome<3:0> = '1010'; // coproc field, always 0xA
            if PSTATE.EL == EL2 then
                AArch32.TakeUndefInstrException(exception);
            else
                AArch32.TakeHypTrapException(exception);
        if HaveEL(EL3) && !ELUsingAAArch32(EL3) then
            if CPTR_EL3.TFP == '1' then
                AArch64.AdvSIMDFPAccessTrap(EL3);
    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() && !ELUsingAAArch32(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);

Shared Pseudocode Functions
// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = !ELUsingAAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '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 = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';
        if trap then
            if target_el == EL1 && EL2Enabled() && !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();
            end
        end
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', '1011xxxxxxxxxxxxxx', '10100xxxxxxxxxxxxx',
        '01001xxxxxxxxxxxx', '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.

AAArch32.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();
    return;
```

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(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();
// 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(Except_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 = Zerps();

  if exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
    // Trapped MRC/MCR, VMRS on FPSID
    iss<19:17> = instr<7:5>;           // opc2
    iss<16:14> = instr<23:21>;         // opc1
    iss<13:10> = instr<19:16>;         // CRn
    iss<8:5>   = instr<15:12>;         // Rt
  elsif 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
  elsif 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';
  elsif exception.exceptype == Exception_Uncategorized then
    // Trapped for unknown reason
    iss<8:5> = instr<19:16>;         // Rn
    iss<3>   = '0';

  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

// 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

// 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

// AArch32.TakeUndefInstrException()
// =================================

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

```cpp
// AArch32.UnderfinedFault()
// ========================
AArch32.UndefinedFault()

if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();
AAArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/functions/aborts/AArch32.CreateFaultRecord

```cpp
// 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 &&
AAArch32.DomainValid(statuscode, level)) then
    fault.domain = domain;
else
    fault.domain = bits(4) UNKNOWN;
fault.debugmoe = debugmoe;
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

```cpp
// 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
    fsr<11> = 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
    fsr<11> = 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 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_Alignment
     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
  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);
Enumeration SRTyp {SRTyp_LSL, SRTyp_LSR, SRTyp_ASR, SRTyp_ROR, SRTyp_RRX};

Library pseudocode for aarch32/functions/common/Shift

// Shift()
// ========

bits(N) Shift(bits(N) value, SRTyp srtype, integer amount, bit carry_in)
  (result, -) = Shift_C(value, srtype, amount, carry_in);
return result;

Library pseudocode for aarch32/functions/common/Shift_C

// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRTyp srtype, integer amount, bit carry_in)
  assert !(srtype == SRTyp_RRX && amount != 1);
  if amount == 0 then
    (result, carry_out) = (value, carry_in);
  else
    case srtype of
      when SRTyp_LSL
        (result, carry_out) = LSL_C(value, amount);
      when SRTyp_LSR
        (result, carry_out) = LSR_C(value, amount);
      when SRTyp_ASR
        (result, carry_out) = ASR_C(value, amount);
      when SRTyp_ROR
        (result, carry_out) = ROR_C(value, amount);
      when SRTyp_RRX
        (result, carry_out) = R RX_C(value, carry_in);
    return (result, carry_out);

Library pseudocode for aarch32/functions/common/T32ExpandImm

// 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()
// ===============
(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)

    if imm12<11:10> == '00' then
        imm32 = ZeroExtend(imm12<7:0>, 32);
    when '01'
        imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
    when '10'
        imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
    when '11'
        imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
    carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value,UInt(imm12<11:7>));
    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)

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if PSTATE.EL == EL0 && !ELUsingAArch32(EL2) then
            return AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);
    // Check for MCR, MRC 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;
    // 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;
    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)

    acctype = 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 = (address == Align(address, size));
    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;
    end if

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
        MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
        AArch32.MarkExclusiveVA(address, ProcessorID(), size);
    end if

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];
    end for

    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

```c
// 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

```c
// FPHalvedSub()
// ==============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FP_RoundingMode(fpcr);
    (type1,sign1,value1) = FP_Unpack(op1, fpcr);
    (type2,sign2,value2) = FP_Unpack(op2, fpcr);
    (done,result) = FP_ProcessNaNs(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 = FP_DefaultNaN();
            FP_ProcessException(FP_Exc_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 != 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;
        endif
    endif
```

Library pseudocode for aarch32/functions/float/FPRSqrtStep

```c
// FPRSqrtStep()
// =============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FP_Unpack(op1, fpcr);
    (type2,sign2,value2) = FP_Unpack(op2, fpcr);
    (done,result) = FP_ProcessNaNs(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()
return '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '0000000000000000000';

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_ATOMICRW, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMICRW };
ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };
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;
// 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;
  bits(size*8) value;
  iswrite = FALSE;

  // MMU or MPU
  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
  transactional = TSTATE.depth > 0 && !(acctype IN {AccType_IFETCH, AccType_PTW});
  accdesc = CreateAccessDescriptor(acctype, transactional);
  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.TagCheckFail(ZeroExtend(address, 64), iswrite);
  value = _Mem[memaddrdesc, size, accdesc];
  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;

  // MMU or MPU
  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
  transactional = TSTATE.depth > 0;
  accdesc = CreateAccessDescriptor(acctype, transactional);
  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.TagCheckFail(ZeroExtend(address, 64), 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;
// 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() 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() 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)
  route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
if !route_to_aarch64 && HaveEl(EL3) && !ElUsingAArch32(EL3)
  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;
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;
mask_set = PSTATE.A == '1';
(-, el) = ElFromM32(target);
intdis = Halted() || ExternalDebugInterruptsDisabled(el);
masked = intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending
if IsPhysicalSErrorPending() && masked then
  syndrome32 = AArch32.PhysicalSErrorSyndrome();
  DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
  ClearPendingPhysicalSError();
return;

Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();
// 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  
syndrome<5:0> = '010001';  // STATUS  
else  
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/functionsregisters/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/functionsregisters/AArch32.ResetSIMDFPRegisters

```c
// AArch32.ResetSIMDFPRegisters()
// ==============================
AArch32.ResetSIMDFPRegisters()
for i = 0 to 15
    Q[i] = bits(128) UNKNOWN;
return;
```

Library pseudocode for aarch32/functionsregisters/AArch32.ResetSpecialRegisters

```c
// AArch32.ResetSpecialRegisters()
// ===============================
AArch32.ResetSpecialRegisters()
// AArch32 special registers
    SPSR_fiq = bits(32) UNKNOWN;
    SPSR_irq = bits(32) UNKNOWN;
    SPSR_svc = bits(32) UNKNOWN;
    SPSR_abt = bits(32) UNKNOWN;
    SPSR_und = 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/functionsregisters/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 UNPREDICTABLE; // UNDEFINED or NOP
    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' && ConstrantUnpredictableBool(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

```plaintext
// 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

```plaintext
bits(32) SP_mon;
bits(32) LR_mon;
```

Library pseudocode for aarch32/functions/registers/PC

```plaintext
// PC - non-assignment form
// ==============

bits(32) PC
return R[15]; // This includes the offset from AArch32 state
```

Library pseudocode for aarch32/functions/registers/PCStoreValue

```plaintext
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
return PC;
```

Library pseudocode for aarch32/functions/registers/Q

```plaintext
// 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()
// -----------

decimal RBankSelect(decimal mode, integer usr, integer fiq, integer irq,
    integer svc, integer abt, integer und, 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/functions/registers/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 CONSTRANGED 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() & ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;
        return;

Library pseudocode for aarch32/functions/registers/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/AArch32.ExecutingATS1xPInstr

// AArch32.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1CPR/WP

boolean AArch32.ExecutingATS1xPInstr()
if !HavePrivATExt() then return FALSE;

  instr = ThisInstr();
  if instr<24+:4> == '1110' && instr<8+:4> == '1110' then
    op1 = instr<21+:3>;
    CRn = instr<16+:4>;
    CRm = instr<0+:4>;
    op2 = instr<5+:3>;
    return (op1 == '000' && CRn == '0111' && CRm == '1001' && op2 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);
```
// AArch32.SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

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 // DBGDSRint
        return TRUE;
    return FALSE;
}

// Write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, bits(32) val);

// Write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, bits(64) val);

// 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;
}
Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

// 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 && HaveEL(EL2) && EL1 && SCR.NS == '1' && HCR.TGE == '1' then
    valid = FALSE;
  if !valid then
    PSTATE.IL = '1';
  else
    AArch32.WriteModeByInstr(model);

Library pseudocode for aarch32/functions/system/BadMode

// 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, M32IRQ, 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;
Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to register
// 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;
// CPSRWriteByInstr()
// ==================
// Update PSTATE.<N,Z,C,V,Q,A,F,M> from a CPSR value written by an MSR instruction.
CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
privileged = PSTATE.EL != EL0;  // PSTATE.<A,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
    // Bit <23> is RES0
    if privileged then
        PSTATE.PAN = value<22>;  // Bits <20> are RES0
        PSTATE.GE = value<19:16>
        PSTATE.DIT = value<21>
    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;  // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>
    if bytemask<0> == '1' then
        // AArch32.WriteModeByInstr() sets PSTATE.I<4:0> = 1 if this is an illegal mode change.
        AArch32.WriteModeByInstr(value<4:0>

return;

// ConditionPassed()
// ================
boolean ConditionPassed()
return ConditionHolds(AArch32.CurrentCond());

// CurrentCond()
bits(4) AArch32.CurrentCond();

// InITBlock()
// ============
boolean InITBlock()
if CurrentInstrSet() == InstrSet_T32 then
    return PSTATE.IT<3:0> != '0000' else
    return FALSE;

// LastInITBlock()
// =---------------
boolean LastInITBlock()
return (PSTATE.IT<3:0> == '1000');
Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

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)

    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.result.fault = AArch32.NoFault();
        if s2desc.memattrs.memtype == MemType_Device ||
            (apply_fore_writeback && s2desc.memattrs.memtype == MemType_Device) then
            result.memattrs.memtype = MemType_Device;
        else
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device, s2desc.memattrs.device);
        end
        if s1desc.memattrs.memtype == MemType_Normal then
            result.memattrs.device = s1desc.memattrs.device;
        elsif s2desc.memattrs.memtype == MemType_Normal then
            result.memattrs.device = s1desc.memattrs.device;
        else                    // Both Device
            result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device, s2desc.memattrs.device);
        end
        result.memattrs.tagged = FALSE;
    end

    result.memattrs = MemAttrDefaults(result.memattrs);

    return result;
MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX, bit C, bit B, bit S, AccType acctype)

    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) = ConstrainingUnpredictableBits(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();
    end;

    // 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;
Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

// AArch32.RemappedTEXDecode()
// ===========================
MemoryAttributes AArch32.RemappedTEXDecode(bits(3) 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_RESPRRR);
            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;
        // transient bits are not supported in this format
        memattrs.inner.transient = FALSE;
        memattrs.outer.transient = FALSE;
        memattrs.tagged = FALSE;
    return MemAttrDefaults(memattrs);
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';
        memattrs.outershareable = SH == '10';
    elsif HaveMTEExt() && attrfield == '11110000' then // Normal, Tagged if 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 = (memattrs.inner.attrs == MemAttr_WB &&
            memattrs.inner.hints == MemHint_RWA &&
            memattrs.outer.attrs == MemAttr_WB &&
            memattrs.outer.hints == MemHint_RWA);
    else
        Unreachable();                          // Reserved, handled above
    return MemAttrDefaults(memattrs);
// 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;
    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;
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = 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.tagged = FALSE;
    end if
    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();
    return result;
// AArch32.AccessIsPrivileged()
// =============

boolean AArch32.AccessIsPrivileged(AccType acctype)
{
    el = AArch32.AccessUsesEL(acctype);
    if (el == EL0)
        ispriv = FALSE;
    else if (el != EL1)
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
}

// 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)
        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') // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESDACR);

    if (attrfield == '00')
        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.AccessIsPrivileged(acctype);

pan = if HavePANExt() then PSTATE.PAN else '0';

is_ldst = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
is_atslxp = (acctype == AccType_AT && AArch32.ExecutingATS1xPInstr());
if pan == '1' && user_r && ispriv && (is_ldst || is_atslxp) then
  priv_r = FALSE;
  priv_w = FALSE;
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));
if ispriv then
  (r, w, xn) = (priv_r, priv_w, priv_xn);
else
  (r, w, xn) = (user_r, user_w, user_xn);

if iswrite then
  fail = xn;
else
  failedread = TRUE;

if acctype == AccType_IFETCH then
  fail = xn;
elseif acctype IN {AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW} then
  fail = !r || !w;
elseif acctype == AccType_DC then
  // DC maintenance instructions operating by VA, cannot fault from stage 1 translation.
  fail = FALSE;
elseif iswrite then
  fail = !w;
else
  fail = !r;

if fail then
  secondstage = FALSE;
s2fs1walk = FALSE;
ipaddress = bits(40) UNKNOWN;
return AArch32.PermissionFault(ipaddress, domain, level, acctype,
!failedread, secondstage, s2fs1walk);
else
    return AArch32.NoFault();

Library pseudocode for aarch32/translation/checks/AArch32.CheckS2Permission

// AArch32.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch32 stage 2 translations

FaultRecord AArch32.CheckS2Permission(Permissions perms, bits(32) vaddress, bits(40) ipaddress,
    integer level, AccType acctype, boolean iswrite,
    boolean s2fs1walk)

assert 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 = !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();

Shared Pseudocode Functions
// 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();

// 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() && DBGSCRext.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.VCRMatch(vaddress);
if size == 4 && !match && AArch32.VCRMatch(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());
match = FALSE;
ispriv = AArch32.AccessIsPrivileged(acctype);
for i = 0 to UInt(DBGDIDR.WRPs)
    match = match || AArch32.WatchpointMatch(i, vaddress, size, ispriv, iswrite);
if match && HaltOnBreakpointOrWatchpoint() then
    reason = DebugHalt_Watchpoint;
    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;
errortype = 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

```plaintext
// 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;
   errortype = 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

```plaintext
// 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;
   errortype = 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

```plaintext
// 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;
   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 acctype, 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';

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, acctype, iswrite, secondstage, 
            s2fs1walk, size);
        permissioncheck = TRUE;  domaincheck = FALSE;
    else
        S1 = AArch32.TranslationTableWalkSD(vaddress, acctype, iswrite, size);
        permissioncheck = TRUE;  domaincheck = TRUE;
    else
        S1 = AArch32.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;  domaincheck = FALSE;
if !IsFault(S1.addrdesc) && UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then
    nTLSMD = if S1TranslationRegime() == EL2 then HSCTLR.nTLSMD else SCTLR.nTLSMD;
    if nTLSMD == '0' then
        S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && acctype != AccType_IFETCH && (acctype == AccType_DCZVA) then
        S1.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && domaincheck 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);
if !IsFault(S1.addrdesc) && 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) && !((HaveNV2Ext() && acctype == AccType_NV2REGISTER) && HasS2Translation()) then
        s2fs1walk = FALSE;
        result = AArch32.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk, size);
    else
        result = S1;
    return result;


// AArch32.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 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);

  assert !IsFault(S2.addrdesc);
  S2.addrdesc.fault = AArch32.AlignmentFault(acctype, iswrite, secondstage);

  if !s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.memtype == MemType_Device
    && HCR.PTW == '1' && S2.addrdesc.memattrs.memtype == MemType_Device then
    domain = bits(4) UNKNOWN;
    S2.addrdesc.fault = AArch32.PermissionFault(ipaddress, domain, S2.level, acctype,
                                               iswrite, secondstage, s2fs1walk);

  result = AArch32.CombineS1S2Desc(S1, S2.addrdesc);
else
  result = S1;
return result;
Library pseudocode for aarch32/translation/translation/AArch32.SecondStageWalk

// AArch32.SecondStageWalk()
// ================
// Perform a stage 2 translation on a stage 1 translation page 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);

Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

// 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 !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
    result.fault = AArch32.CheckDebug(vaddress, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(vaddress);

return result;
Library pseudocode for aarch32/translation/walk/AArch32.TranslationTableWalkLD
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

pseudocode. 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 s2fslwalk, 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

domain = bits(4) UNKNOWN;
descaddr.memattrs.metype = MemType_Normal;

---

// Fixed parameters for the page 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 page 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);
    if el == EL2 then
        inputsize = 32 - UInt(HTCR.T0SZ);
        basefound = inputsize == 32 || IsZero(inputaddr<31:inputsize>);
        disabled = FALSE;
        baseregister = HTTBR;
        descaddr.memattrs = WalkAttrDecode(HTCR.SH0, HTCR.ORGN0, HTCR.IRGN0, secondstage);
        reversedescriptors = HSCTLR.EE == '1';
        lookupsecure = FALSE;
        singlepriv = TRUE;
        hierattrsdidabes = 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);
            hierattrsdidabes = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD0 == '1';
        tsize = UInt(TTBCR.T1SZ);
        if (tsize == 0 && !basefound) || (tsize > 0 && IsOnes(inputaddr<31:(32-tsize)>) then
            inputsize = 32 - tsize;
            basefound = TRUE;
            baseregister = TTBR1;
            descaddr.memattrs = WalkAttrDecode(TTBCR.SH1, TTBCR.ORGN1, TTBCR.IRGN1, secondstage);
            hierattrsdidabes = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD1 == '1';
            reversedescriptors = SCTLR.EE == '1';
        lookupsecure = IsSecure();
    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);
            hierattrsdidabes = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD0 == '1';
            reversedescriptors = HSCTLR.EE == '1';
        lookupsecure = FALSE;
    end
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);
        hierattrsdidabes = AArch32.HaveHPDExt() && TTBCR.T2E == '1' && TTBCR2.HPD0 == '1';
        reversedescriptors = HSCTLR.EE == '1';
    lookupsecure = FALSE;
end

---

Shared Pseudocode Functions
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_RESVTCR);
basefound = inputsize == 40 || IsZero(inputaddr<39:inputsize>);
disabled = FALSE;
descaddr.memattrs = WalkAttrDecode(VTCR.SH0, VTCR.ORGN0, VTCR.IRGN0, 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)
startsizecheck = inputsize - ((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)
// 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(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(Number 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() || (HaveNV2Ext() && acctype == AccType_NV2REGISTER) 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;

// 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);
acccdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
desc = _Mem[descaddr2, 8, accdesc];
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
    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 && !hierattrsdisabled 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 = desc<11>;
sh = desc<9:8>;
memattr = desc<5:2>;
// AttrIndx and NS bit in stage 1

Shared Pseudocode Functions
result.domain = bits(4) UNKNOWN;                             // Domains not used
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
    // PXN, nG and AP[1] apply only in EL1&0 stage 1 translation regimes
    if singlelepriv then
        result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);       // Force privileged only
        result.perms.pxn   = pxn OR pxn_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
    else
        result.perms.ap<1> = '1';
        result.perms.xn   = '0';
        result.nG          = '0';
    end
else
    result.perms.ap<2:1> = ap<2:1>;                            // Stage 1 block or pages might be guarded
    result.perms.ap<0>   = '1';
    result.perms.xn      = xn;
    if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
    result.perms.pxn     = '0';
    result.nG            = '0';
    if s2fslwalk then
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_PTW);
    else
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
    end
    result.addrdesc.paddress.NS = '1';
end
else
    result.addrdesc.paddress.address = ZeroExtend(outputaddress);
end
result.addrdesc.fault = AArch32.NoFault();
result.contiguous = contiguousbit == '1';
if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;

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
domain = bits(4) UNKNOWN;

// Determine correct Translation Table Base Register to use.
bits(64) ttbr;
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 this Translation Table Base Register is not disabled.
if disabled then
  level = 1;
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 !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) 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 = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
l1desc = _Mem[l1descaddr2, 4, accdesc];
if SCTLR.EE == '1' then l1desc = BigEndianReverse(l1desc);
// Process descriptor from initial lookup.
case l1desc<1:0> of
  when '00' // Fault, Reserved
    level = 1;
    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.
    l2descadr.paddress.address = ZeroExtend(l1desc<31:10>:vaddress<19:12>:'00');
    l2descadr.paddress.NS = if IsSecure() then '0' else '1';
    l2descadr.memattrs = l1descadr.memattrs;
    if !HaveEL(EL2) || (IsSecure() && !IsSecureEL2Enabled()) then
      // if only 1 stage of translation
      l2descadr2 = l2descadr;
    else
      l2descadr2 = AArch32.SecondStageWalk(l2descadr, vaddress, acctype, iswrite, 4);
      // Check for a fault on the stage 2 walk
      if IsFault(l2descadr2) then
        result.addrdesc.fault = l2descadr2.fault;
        return result;
    // Update virtual address for abort functions
    l2descadr2.vaddress = ZeroExtend(vaddress);
    accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
    l2desc = _Mem[l2descadr2, 4, accdesc];
    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;
    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;
    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>);
  when '1x' // Section or Supersection
if SCTLR.AFE == '1' && l1desc<10> == '0' then
    // Only bit 10 used to support ARM System Mode
    result.addrdesc.fault = AArch32.AccessFlagFault(ipaddress, domain, level, acctype, iswrite, secondstage, s2fs1walk);
    return result;

if l1desc<18> == '0' then                              // Section
    domain = l1desc<8:5>;
    blocksize = 1024;
    outputaddress = ZeroExtend(l1desc<31:20>:vaddress<19:0>);
else                                               // Supersection
    domain = '0000';
    blocksize = 16384;
    outputaddress = l1desc<8:5>:l1desc<23:20>:l1desc<31:24>:vaddress<23:0>;

// 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.nG = nG;
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

boolean RemapRegsHaveResetValues();
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)
    assert !ELUsingAAArch32($1TranslationRegime());
    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
// CONSTRAINED 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 CONSTRAINED
// UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
// at the address DBGVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    match = value_match && state_match && enabled;

return match;
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()) ||              // Context matching
        dbgtype == '010x' ||                                             // Reserved
        (dbgtype != '0x0x' && !context_aware) ||                         // Context matching
        (dbgtype == '1xxx' && !HaveEL(EL2))) then                        // EL2 extension
        (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_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                     // "vaddress" is word aligned
            byte_select_match = TRUE;             // DBGBCR_EL1[n].BAS<byte> is RES1
        top = AddrTop(vaddress, TRUE, PSTATE.EL);
        BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> & & byte_select_match;
    elsif match_cid then
        if IsInHost() then
            BVR_match = (CONTEXTIDR_EL2 == DBGBVR_EL1[n]<31:0>);
        else
            BVR_match = (PSTATE.EL IN {EL0, EL1} & & CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
        elsif match_cid1 then
            BVR_match = (PSTATE.EL IN {EL0, EL1} & & !IsInHost() & & CONTEXTIDR_EL1 == DBGBVR_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(DBGBVR_EL1[n]<39:32>, 16);
  else
    vmid = VTTBR_EL2.VMID;
    bvr_vmid = DBGBVR_EL1[n]<47:32>;
  BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                  !IsInHost() &&
                  vmid == bvr_vmid);
elsif match_cid2 then
  BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                 DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2);
end if

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;
Library pseudocode for aarch64/debug/breakpoint/AArch64.StateMatch

// AArch64.StateMatch()
// ===========
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

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 == Constraint_DISABLED then return FALSE;
else EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
EL2_match = HaveEL(EL2) && (HMC == '1' && (SSC:PxC != '1000')) || SSC == '11';
EL1_match = PxC<0> == '1';
EL0_match = PxC<1> == '1';

if HaveNV2Ext() && acctype == AccType_NV2REGISTER && !isbreakpnt then
    priv_match = EL2_match;
elsif !ispriv && !isbreakpnt then
    priv_match = EL0_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;
    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.
        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 (Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN); case c of
            when Constraint_DISABLED return FALSE;            // Disabled
            when Constraint_NONE linked = FALSE;              // No linking
            end_case // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint
        if linked then
            vaddress = bits(64) UNKNOWN;
            linked_to = TRUE;
            linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);
        end_if
    end_if

    return priv_match && security_state_match && (!linked || linked_match);
Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```java
// AArch64.GenerateDebugExceptions()
// ---------------------------------

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```java
// AArch64.GenerateDebugExceptionsFrom()
// -------------------------------------

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)
    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;
    route_to_el2 = HaveEL2() && (!secure || IsSecureEL2Enabled()) && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);
    enabled = !HaveEL3() || !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

```java
// AArch64.CheckForPMUOverflow()
// -----------------------------
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()
    pmuiq = PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSET_EL0<31> == '1';
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL() then
            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' && PMOVSET_EL0<n> == '1' then pmuiq = TRUE;
    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuiq then HIGH else LOW);
    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuiq 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 pmuiq;
```
library pseudocode for aarch64/debug/pmu/AArch64.CountEvents

// AArch64.CountEvents()
// =====================
// Return TRUE if counter “n” should count its event. For the cycle counter, n == 31.

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
        E = if n < UInt(MDCR_EL2.HPMN) || n == 31 then PMCR_EL0.E else MDCR_EL2.HPME;
    else
        E = PMCR_EL0.E;
    enabled = E == '1' && PMCNTENSET_EL0<n> == '1';
    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    prohibited = HaveEL(EL3) && IsSecure() && MDCR_EL3.SPME == '0';
    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * Executing at EL2
    // * PMN<31> is not reserved for EL2
    // * MDCR_EL2.HPMD == 1
    if !prohibited && HaveEL(EL2) && HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(MDCR_EL2.HPMN) || n == 31) then
        prohibited = (MDCR_EL2.HPMD == '1');
    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();
    // For the cycle counter, PMCR_EL0.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR_EL0.DP == '1');
    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} 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';
    M = if HaveEL(EL3) then filter<26> else '0';
    SH = if 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;
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 PMSCR_EL1.CX == '1';

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

// CollectContextIDR2()
// --------------------------
boolean CollectContextIDR2()
if !StatisticalProfilingEnabled() then return FALSE;
if EL2Enabled() then return FALSE;
return PMSCR_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) then
    return PMSCR_EL2.PA == '1' && (el == EL2 || PMSCR_EL1.PA == '1');
else
    return PMSCR_EL1.PA == '1';
Library pseudocode for aarch64/debug/statisticalprofiling/CollectRecord

// CollectRecord()
// =============

boolean CollectRecord(bits(64) events, integer total_latency, OpType optype)
assert StatisticalProfilingEnabled();

// Filtering by event
if PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
    bits(64) mask = 0xFFFF0000FF00F0AA<63:0>; // Bits [63:48,31:24,15:12,7,5,3,1]
if HaveStatisticalProfiling() then
    mask<11> = '1'; // Alignment flag
if HaveSVE() then mask<18:17> = Ones(); // Predicate flags
e = events AND mask;
m = PMSEVFR_EL1 AND mask;
if !IsZero(NOT(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)) ||
    (PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
    (PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT))) then
    return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
return TRUE;

Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

// CollectTimeStamp()
// ================

TimeStamp CollectTimeStamp()
assert StatisticalProfilingEnabled();
(TimeStamp_None)
if (secure, el) = ProfilingBufferOwner();
if el == EL2 then
    if PMSCR_EL2.TS == '0' then return TimeStamp_None;
else
    if PMSCR_EL1.TS == '0' then return TimeStamp_None;
if EL2Enabled() then
    pct = PMSCR_EL2.PCT == '01' && (el == EL2 || PMSCR_EL1.PCT == '01');
else
    pct = PMSCR_EL1.PCT == '01';
return (if pct then TimeStamp_Physical else TimeStamp_Virtual);
Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```c
enum 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

// 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/StatisticalProfilingEnabled

```c
// 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

```c
enum SysRegAccess { SysRegAccess_OK,
                  SysRegAccess_UNDEFINED,
                  SysRegAccess_TrapToEL1,
                  SysRegAccess_TrapToEL2,
                  SysRegAccess_TrapToEL3 };
```

Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```c
enum TimeStamp { TimeStamp_None, // No timestamp
               TimeStamp_CoreSight, // CoreSight time (IMPLEMENTATION DEFINED)
               TimeStamp_Virtual,   // Physical counter value minus CNTVOFF_EL2
               TimeStamp_Physical }; // Physical counter value with no offset
```
// AArch64.TakeExceptionInDebugState()
// ----------------------------------
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
  assert HaveEL(target_el) && !ELUsingAAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
  sync_errors = HaveIESB() && SCTLR[].IESB == '1';
  if HaveDoubleFaultExt() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
  // SCTLR[].IESB might be ignored in Debug state.
  if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
  if 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();
    MaybeZeroSVEUppers(target_el);

  AArch64.ReportException(exception, target_el);

  PSTATE.EL = target_el;
  PSTATE.nRW = '0';
  PSTATE.SP = '1';

  SPSR[] = bits(32) 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.SP = '0';

  if from_32 then // Coming from AArch32
    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';

  DLR_EL0 = bits(64) UNKNOWN;
  DSPSR_EL0 = bits(32) UNKNOWN;

  EDSCR.ERR = '1';
  UpdateEDSCRFIELDS();

  if sync_errors then
    SynchronizeErrors();

  EndOfInstruction();
// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n, AccType accctype, bits(64) vaddress)

    el = if HaveNV2Ext() && accctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
    top = AddrTop(vaddress, FALSE, el);
    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_WPBASCONTIGUOUS);
            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
            end case
            if mask > bottom then
                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_WPMASKEDBITS);
                else
                    WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;
                end if
            end if
        end if
    end if
    return WVR_match && byte_select_match;
Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

// 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(S1TranslationRegime());
assert n <= UInt(ID_AA64DFR0_EL1.WRPs);

// "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all
// load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
// loads.
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);

ls_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 && ls_match && enabled;

Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

// 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(Exceptype exceptype, FaultRecord fault, bits(64) vaddress)
    exception = ExceptionSyndrome(exceptype);
    d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_NV2Watchpoint};
    exception.syndrome = 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.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
    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/aborts/AArch64.EffectiveTCF

```plaintext
// AArch64.EffectiveTCF()
// Returns the TCF field applied to Tag Check Fails 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;
    if tcf == '11' then
        (-,tcf) = ConstrainUnpredictableBits(Unpredictable_RESTCF);
    return tcf;
```

### Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```plaintext
// AArch64.InstructionAbort()
// External aborts on instruction fetch must be taken synchronously

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();
    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);
    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/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.ReportTagCheckFail

// AArch64.ReportTagCheckFail()
// ============================
// Records a tag fail exception into the appropriate TCFR_ELx.

AArch64.ReportTagCheckFail(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.SPAlignmentFault

// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception.SPAlignment);

    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.TagCheckFail

// AArch64.TagCheckFail()
// ======================
// Handle a tag check fail condition.

AArch64.TagCheckFail(bits(64) vaddress, boolean iswrite)
    bits(2) tcf = AArch64.EffectiveTCF(PSTATE.EL);
    if tcf == '01' then
        AArch64.TagCheckFault(vaddress, iswrite);
    elsif tcf == '10' then
        AArch64.ReportTagCheckFail(PSTATE.EL, vaddress<55>);

library pseudocode for aarch64/exceptions/aborts/AArch64.TagCheckFault

// AArch64.TagCheckFault()
// =======================
// Raise a tag check fail exception.

AArch64.TagCheckFault(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;
    end if
    exception = ExceptionSyndrome(Exception_DataAbort);
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';
    end if
    exception.vaddress = va;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

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);
// 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.IMO == '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);
elsif 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);

// AArch64.TakePhysicalIRQException()
// ---------------------------------

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);
elsif 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 = HaveEL(EL3) && SCR_EL3.EA == '1';
route_to_el2 = (PSTATE.EL IN {EL0, EL1}) && EL2Enabled() &&
(SCR_EL2.TGE == '1' || (!isInHost() && HCR_EL2.AMO == '1'));

bits(64) preferred_exception_return = ThisInstrAddr();
vec_offset = 0x180;

exception = ExceptionSyndrome(Exception_SError);
exception.syndrome<24> = if impdef_syndrome then '1' else '0';
exception.syndrome<23:0> = syndrome;

ClearPendingPhysicalSError();

if PSTATE.EL == EL3 || route_to_el3 then
    AArch64.TakeException(EL3, exception, preferred_exception_return, vec_offset);
elsif PSTATE.EL == EL2 || route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vec_offset);
else
    AArch64.TakeException(EL1, exception, preferred_exception_return, vec_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();
vec_offset = 0x100;

exception = ExceptionSyndrome(Exception_FIQ);

AArch64.TakeException(EL1, exception, preferred_exception_return, vec_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();
vec_offset = 0x80;

exception = ExceptionSyndrome(Exception_IRQ);

AArch64.TakeException(EL1, exception, preferred_exception_return, vec_offset);
Library pseudocode for aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

// 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

// 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

// 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();
    vect_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';
    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 != EL2;
    assert 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_VectorCatch, fault, vaddress);
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);

Library pseudocode for aarch64/exceptions/debug/AArch64.WatchpointException

// 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 exceptype, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1';          // AArch64 instructions always 32-bit

    case exceptype of
        when Exception_Uncategorized       ec = 0x00; il = '1';
        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_CP14DRTTrap         ec = 0x06;          assert from_32;
        when Exception_AdvSIMDFPAccessTrap ec = 0x07;
        when Exception_FPIDTrap            ec = 0x08;
        when Exception_PACTrap             ec = 0x09;
        when Exception_TSTARTAccessTrap    ec = 0x1B;
        when Exception_CP14RRTTrap         ec = 0x0C;          assert from_32;
        when Exception_BranchTarget        ec = 0x0D;
        when Exception_IllegalState        ec = 0x0E; il = '1';
        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_SVEAccessTrap       ec = 0x19;          assert !from_32;
        when Exception_ERetTrap            ec = 0x1A;
        when Exception_PACFail             ec = 0x1C;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment         ec = 0x22; il = '1';
        when Exception_DataAbort           ec = 0x24;
        when Exception_NV2DataAbort        ec = 0x25;
        when Exception_SPAIignment         ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPtrappedException  ec = 0x28;
        when Exception_SError              ec = 0x2F; il = '1';
        when Exception_Breakpoint          ec = 0x30; il = '1';
        when Exception_SoftwareStep        ec = 0x32; il = '1';
        when Exception_Watchpoint          ec = 0x34; il = '1';
        when Exception_NV2Watchpoint       ec = 0x35; il = '1';
        when Exception_SoftwareBreakpoint  ec = 0x38;
        when Exception_VectorCatch         ec = 0x3A; il = '1'; 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;

    return (ec,il);
// 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;

  // 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] = ec<5:0>:il:iss;

  if exceptype IN {
      Exception/InstructionAbort, Exception_PCAlignment, 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.ipavvalid then
          HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
          if HaveSecureEL2Ext() then
              if IsSecureEL2Enabled() then
                  HPFAR_EL2.NS = exception.NS;
              else
                  HPFAR_EL2.NS = '0';
          else
              HPFAR_EL2.NS = 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);
Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

// 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;

    // 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.IL = '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;

    // 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

```cpp
AArch64.FPTrappedException()
// ============================
AArch64.FPTrappedException(boolean is_ase, integer element, 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
        end if
    else
        exception.syndrome<23> = '1'; // TFV
    end if
    exception.syndrome<10:8> = bits(3) UNKNOWN;
    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;
    end if
    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);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
    end if
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

```cpp
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);
    end if
```
Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```plaintext
// 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

```plaintext
// 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);
```
Library pseudocode for aarch64/exceptions/takeexception/AArch64.TakeException
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() && SCLTR[].IESB == '1';
if HaveDoubleFaultExt() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
if sync_errors && InsertIESBBeforeException(target_el) then
    SynchronizeErrors();
    iesb_req = FALSE;
    sync_errors = FALSE;
TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

if 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 = UsingAArch32();
if from_32 then
    MaybeZeroRegisterUppers(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);
    elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
        lower_32 = ELUsingAArch32(EL0);
    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;
spsr = GetPSRFromPSTATE();
if HaveNVExt() && PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() && HCR_EL2.<NV,NV1> == '10' then
    spsr<3:2> = '10';
if HaveBTIExt() then
    // SPSR[].BTYPE is only guaranteed valid for these exception types
    if exception.exctype IN {Exception_SError, Exception_IRQ, Exception_FIQ, Exception_SoftwareStep, Exception_PCAlignment, ExceptionInstructionAbort, 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.exctype == Exception_NV2DataAbort && target_el == EL3 then
    // external aborts are configured to be taken to EL3
    exception.exctype = Exception_DataAbort;
if !(exception.exctype IN {Exception_IRQ, Exception_FIQ}) then
AArch64.ReportException(exception, target_el);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SP = '0';
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';
BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

if sync_errors then
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterruptions(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()
// ------------------------------------------
// 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>;
    endif
else
    iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
end;

elsif 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>;
    endif
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;

elsif 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;
    end;
    iss<3>   = '1';
elsif 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';
endif

iss<0> = instr<20>;          // Direction

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);
    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;
    // 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 {0,4, 8 }) ||
         (CRn == 11 & CRm IN {0,1,2,3,4,5,6,7,8,15})) then
        return TRUE;
    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
        case CPACR[].FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);
        AArch64.CheckFPAdvSIMDEnabled(); // 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 = !(PSTATE.EL == EL1 & HCR_EL2.TGE == '1');
                when '01' disabled = (PSTATE.EL == EL0 & HCR_EL2.TGE == '1');
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);
        endif
    endif

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
    endif

    return;

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETTA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETTA, 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 = ExceptionSyndrome(Exception_ERetTrap);
        if eret_with_pac then // ERET
            exception.syndrome<1> = '0';
            exception.syndrome<0> = '0'; // RES0
        else
            exception.syndrome<1> = '1';
            if pac_uses_key_a then // ERETTA
                exception.syndrome<0> = '0';
            else // ERETAB
                exception.syndrome<0> = '1';
            endif
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    endif
Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
if PSTATE.EL == EL0 then UNDEFINED;
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' &&
  if route_to_el2 then
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;
    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);
// 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);

// 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);

// 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 // Trapped access due to unknown reason.
            exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7 // Trapped access to SVE, Advance SIMD&FP system register.
            exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
        when 0x18 // Trapped access to system register or system instruction.
            instr = ThisInstr();
            exception.syndrome<21:20> = instr<20:19>; // Op0
            exception.syndrome<19:17> = instr<7:5>; // Op2
            exception.syndrome<16:14> = instr<18:16>; // Op1
            exception.syndrome<13:10> = instr<15:12>; // CRn
            exception.syndrome<9:5> = instr<4:0>; // Rt
            exception.syndrome<4:1> = instr<11:8>; // CRm
            exception.syndrome<0> = instr<21>; // Direction
        when 0x19 // Trapped access to SVE System register
            exception = ExceptionSyndrome(Exception_SVEAccessTrap);
        otherwise
            Unreachable();
    return exception;

Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

// AArch64.UndefinedFault()
// ========================

AArch64.UndefinedFault()

    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

```c
// 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 = Exception Syndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = Condition Syndrome();
    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

```c
// CheckFPAdvSIMDEnabled64()
// ================

// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

Library pseudocode for aarch64/functions/aborts/AArch64.CreateFaultRecord

```c
// 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;
```
Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception Level using AArch64.

bits(25) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;
    bits(25) iss = Zeros();
    if HaveRASExt() && IsExternalSyncAbort(fault) then iss<12:11> = fault.errortype; // SET
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
            iss<13> = '1'; // Value of '1' indicates fault is generated by use of VNCR_EL2
        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;

Library pseudocode for aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

// AArch64.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 AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)
    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);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    if passed 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/AAArch64.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);

Library pseudocode for aarch64/functions/exclusive/AAArch64.SetExclusiveMonitors

// 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 = (address == Align(address, size));
    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);
Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

// FPRSqrtStepFused()
// ===============

bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
assert N IN {16, 32, 64};
bits(N) result;
op1 = FPNeg(op1);
(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 = 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 FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
      result = FPZero(sign);
    else
      result = FPRound(result_value, FPCR);
  end
return result;

Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

// FPRecipStepFused()
// ===============

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
assert N IN {16, 32, 64};
bits(N) result;
op1 = FPNeg(op1);
(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('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 FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
      result = FPZero(sign);
    else
      result = FPRound(result_value, FPCR);
  end
return result;
Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

```java
// 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() then return FALSE;
if acctype IN {AccType_IFETCH, AccType_PTW} then return FALSE;
if acctype == AccType_NV2REGISTER then return FALSE;
if PSTATE.TCO=='1' then return FALSE;
if IsNonTagCheckedInstruction() then return FALSE;
return TRUE;
```

Library pseudocode for aarch64/functions/memory/AArch64.AddressWithAllocationTag

```java
// 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, bits(4) allocation_tag)
bits(64) result = address;
bits(4) tag;
if AArch64.AllocationTagAccessIsEnabled() then
tag = allocation_tag;
else
tag = '0000';
result<59:56> = tag;
return result;
```

Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

```java
// 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>;
```
// 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_ATOMICRW);
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW, AccType_ATOMICRW, AccType_ATOMIC);
    vector = acctype == AccType_VEC;
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveUA16Ext() 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));
    end
    return aligned;
}

// 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;
}
// 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;
      bits(size*8) value;
      iswrite = FALSE;

    // MMU or MPU
    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
    transactional = TSTATE.depth > 0 && !(acctype IN {AccType_IFETCH,AccType_PTW});
    accdesc = CreateAccessDescriptor(acctype, transactional);
    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.TagCheckFail(ZeroExtend(address, 64), iswrite);
        value = _Mem[memaddrdesc, size, accdesc];
    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;

    // MMU or MPU
    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
    transactional = TSTATE.depth > 0;
    accdesc = CreateAccessDescriptor(acctype, transactional);
    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.TagCheckFail(ZeroExtend(address, 64), iswrite);
        _Mem[memaddrdesc, size, accdesc] = value;
    return;
Library pseudocode for aarch64/functions/memory/AArch64.MemTag

```
// 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() && 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);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
if AArch64.AllocationTagAccessIsEnabled() && 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.TagCheckFail(address, iswrite);  
  return memaddrdesc;

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 = (SCTLR[].SA0 != '0');
  else
    stack_align_check = (SCTLR[].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();
// 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> = 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 (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '-1') || BigEndian() 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 (HaveNV2Ext() && acctype == AccType_NV2REGISTER && SCTLR_EL2.EE == '-1') || BigEndian() 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

```pseudocode
// 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];
  if BigEndian() then
    oldvalue = BigEndianReverse(oldvalue);
  end if;
  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;
  end case;
  if BigEndian() then
    newvalue = BigEndianReverse(newvalue);
  end if;
  _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];
    if BigEndian() then
        oldvalue = BigEndianReverse(oldvalue);
    if oldvalue == expectedvalue then
        if BigEndian() then
            newvalue = BigEndianReverse(newvalue);
        _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;
    return oldvalue;
```

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;
  bit selbit;
  boolean tbi = CalculateTBI(ptr, data);
  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>;
      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 selbit = if tbi then ptr<55> else 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 selbit = if tbi then ptr<55> else ptr<63>;
    end
  end

  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
  ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
  if tbi then
    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>);
  end

  // 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);
// 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);
Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

// 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);
elseif 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 = 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;

        if Enable == '0' then return X;
        elsif TrapEL2 then TrapPACUse(EL2);
        elsif TrapEL3 then TrapPACUse(EL3);
        else return AddPAC(X, Y, APIAKey_EL1, FALSE);

    end;
Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

// 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 = STTranslationRegime() == 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 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);
elsif 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 = CalculateTBI(ptr, data);
integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
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 = ptr;
        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
            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);
    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);
    else
        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;
    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 Auth(X, Y, APDAKey_EL1, TRUE, '0', is_combined);
```
// 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;
  endcase

  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;

    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);
    end case;
Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

// CalculateBottomPACBit()
// =======================

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 HaveEL(EL2);
            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';
        end
    else
        // EL2 translation regime registers
        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';
    end

    max_limit_tsz_field = (if !HaveSmallPageTblExt() 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};
        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);
boolean CalculateTBI(bits(64) ptr, boolean data)
    boolean tbi = FALSE;
    if PptrHasUpperAndLowerAddrRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                tbi = if ptr<55> == '1' then TCR_EL1.TBI1 == '1' else TCR_EL1.TBI0 == '1';
            else
                if ptr<55> == '1' then
                    tbi = TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0';
                else
                    tbi = TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0';
            else
                // EL2 translation regime registers
                if data then
                    tbi = if ptr<55> == '1' then TCR_EL2.TBI1 == '1' else TCR_EL2.TBI0 == '1';
                else
                    if ptr<55> == '1' then
                        tbi = TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0';
                    else
                        tbi = TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0';
        else
            // EL3 translation regime registers
            if data then
                tbi = if data then TCR_EL3.TBI=='1' else TCR_EL3.TBI=='1' && TCR_EL3.TBID=='0';
            else
                if data then TCR_EL3.TBI=='1' else TCR_EL3.TBI=='1' && TCR_EL3.TBID=='0';
    return tbi;
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] = 0x13198A2E0E3707344<63:0>;
   RC[2] = 0xA4093822229F31D0<63:0>;
   RC[3] = 0x082EFA98EC46C89<63:0>;
   RC[4] = 0x452821E638001377<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);
         workingval = PACSub(workingval);
         runningmod = TweakShuffle(runningmod<63:0>);
      end if
      roundkey = modk0 EOR runningmod;
      workingval = workingval EOR roundkey;
      workingval = workingval EOR RC[4-i];
      workingval = workingval EOR Alpha;
      workingval = workingval EOR modk0;
   end for
   return workingval;
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<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<63:60>;
    outdata<59:56> = indata<63:60>;
    return outdata;
}
// 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> = '0110';
    when '1001'  Toutput<4*i+3:4*i> = '0111';
    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';
  return Toutput;

// 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+8)+3:4*(i+8)>, 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+8)+3:4*(i+8)>, 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

```c
// 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';
  return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/RotCell

```c
// 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

```c
// 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

```c
// 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> = 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<47:44>);
    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> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = TweakCellRot(indata<51:48>);
    outdata<39:36> = TweakCellRot(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> = 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 en指导意见

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

// 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

// 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

// 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

// 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

// 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)
  boolean TrapEL2;
  boolean TrapEL3;
  bits(64) original_ptr;
  bits(64) extfield;
  boolean tbi = CalculateTBI(A, data);
  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>;
  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 = FALSE;
    when EL3
      TrapEL2 = FALSE;
      TrapEL3 = FALSE;
    if TrapEL2 then TrapPACUse(EL2);
    elsif TrapEL3 then TrapPACUse(EL3);
    else 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);
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
if IsPhysicalSErrorPending() && masked 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)<31:0>;
        ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0

return;

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(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 = vintdis || 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>)<31:0>;
    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;
elsif 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/functionsregisters/AArch64.ResetSIMDFPRegisters

// AArch64.ResetSIMDFPRegisters()
// -----------------------------

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;

Library pseudocode for aarch64/functionsregisters/AArch64.ResetSpecialRegisters

// AArch64.ResetSpecialRegisters()
// --------------------------------

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;
        ELR_EL3 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    return;

Library pseudocode for aarch64/functionsregisters/AArch64.ResetSystemRegisters

AArch64.ResetSystemRegisters(boolean cold_reset);

Library pseudocode for aarch64/functionsregisters/PC

// PC - non-assignment form
// -----------------------------
// Read program counter.

bits(64) PC[]
    return _PC;
// 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);
    end case;
return;

// 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>;
    end case;
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);

// 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);
Library pseudocode for aarch64/functions/sve/AArch32.IsFPEnabled

// AArch32.IsFPEnabled()
//=======================================================

boolean AArch32.IsFPEnabled(bits(2) el)
if el == EL0 & !ELUsingAArch32(EL1) then
    return AArch64.IsFPEnabled(el);

if HaveEL(EL1) & !ELUsingAArch32(EL1) & !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 'x0' disabled = TRUE;
        when '01' disabled = (el == EL0);
        when '11' disabled = FALSE;
    if disabled then return FALSE;

if el IN {EL0, EL1, EL2} then
    if EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            if CPTR_EL2.TFP == '1' then return FALSE;
        else
            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;

Library pseudocode for aarch64/functions/sve/AArch64.IsFPEnabled

// AArch64.IsFPEnabled()
//=======================================================

boolean AArch64.IsFPEnabled(bits(2) el)
// Check if access disabled in CPACR_EL1
if el IN {EL0, EL1} then
    // Check FP&SIMD at EL0/EL1
    case CPACR[].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
        if CPTR_EL2.FPEN == 'x0' 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 >= 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()
// ===============

CheckSVEEnabled()
// Check if access disabled in CPACR_EL1
if PSTATE.EL IN {EL0, EL1} & HCR_EL2.<E2H, TGE> != '11' then
  // Check SVE at EL0/EL1
case CPACR[].ZEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = PSTATE.EL == EL0;
    when '11' disabled = FALSE;
  if disabled then SVEAccessTrap(EL1);

  // Check FP&SIMD at EL0/EL1
case CPACR[].FPEN of
    when 'x0' disabled = TRUE;
    when '01' disabled = PSTATE.EL == EL0;
    when '11' disabled = FALSE;
  if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);
  if PSTATE.EL IN {EL0, EL1, EL2} & EL2Enabled() then
    if HaveVirtHostExt() & HCR_EL2.E2H == '1' then
      if CPTR_EL2.ZEN == 'x0' || (PSTATE.EL == EL0 & HCR_EL2.TGE == '1' & CPTR_EL2.ZEN == '01') then
        SVEAccessTrap(EL2);
      if CPTR_EL2.FPEN == 'x0' || (PSTATE.EL == EL0 & HCR_EL2.TGE == '1' & CPTR_EL2.FPEN == '01') then
        AArch64.AdvSIMDFPAccessTrap(EL2);
      // 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 '11001' numElem = elements - (elements MOD 4); when '11010' 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

```c
// 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);
if type1==FPType_SNaN || type2==FPType_SNaN || type1==FPType_QNaN || type2==FPType_QNaN then
  result = TRUE;
else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
  result = (value1 != value2);
return result;
```

Library pseudocode for aarch64/functions/sve/FPCompareUN

```c
// 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);
return (type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN);
```

Library pseudocode for aarch64/functions/sve/FPConvertSVE

```c
// FPConvertSVE()
// ===============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr, FPRoundingMode rounding)
fpcr.AHP = '0';
return FPConvert(op, fpcr, rounding);
```

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;
```
Library pseudocode for aarch64/functions/sve/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 = 0x0e43a;
      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 = 0x1b8d3a;
      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 = 0x3df08a4;
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 = 0x4be5c15;
when 44 result = 0x4e248c;
when 45 result = 0x506354;
when 46 result = 0x52a81e;
when 47 result = 0x54f35b;
when 48 result = 0x5744fd;
when 49 result = 0x599d16;
when 50 result = 0x5bfb88;
when 51 result = 0x5e60f5;
when 52 result = 0x60ccdf;
when 53 result = 0x633f89;
when 54 result = 0x65b907;
when 55 result = 0x68396a;
when 56 result = 0x6ac0c7;
when 57 result = 0x6d4f30;
when 58 result = 0x6fe4ba;
when 59 result = 0x728177;
when 60 result = 0x75257d;
when 61 result = 0x77d0df;
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 = 0x1429aaa92de9;
    when 8 result = 0x172b83c7d517b;
    when 9 result = 0x1a35beb6fcb75;
    when 10 result = 0x1d487316bb9aa;
    when 11 result = 0x2063bb8628cd6;
    when 12 result = 0x23b7a6e765238;
    when 13 result = 0x26b456e27cd0;
    when 14 result = 0x29e9df518e61;
    when 15 result = 0x2d285a6e4030b;
    when 16 result = 0x306f0a31b715;
    when 17 result = 0x330b26416ff;
    when 18 result = 0x371a7373aa9cb;
    when 19 result = 0x3a7b034e59ff7;
    when 20 result = 0x3dea64c123422;
    when 21 result = 0x4160a21f72e2a;
    when 22 result = 0x44e0860618920;
    when 23 result = 0x486a2b5c13cd0;
    when 24 result = 0x4bfda5362a27;
    when 25 result = 0x4f9b2769d2ca7;
    when 26 result = 0x5342b569d0f82;
    when 27 result = 0x56f473685b7da;
    when 28 result = 0x5ab07dd485429;

Shared Pseudocode Functions

Page 3075
when 29 result = 0x5E76F15AD2148;
when 30 result = 0x6247EB03A5585;
when 31 result = 0x6623882552225;
when 32 result = 0x6A09E667F3BCD;
when 33 result = 0x6DFB23C651A2F;
when 34 result = 0x71F75E8EC5F74;
when 35 result = 0x75FE564267C9;
when 36 result = 0x7A11473EB0187;
when 37 result = 0x7E2F336CF4E62;
when 38 result = 0x82589994CCCE13;
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 = 0xA5503B23E255D;
when 47 result = 0xA9E6B5579FDBF;
when 48 result = 0xAE899F95AD3AD;
when 49 result = 0xB33A2B84F15FB;
when 50 result = 0xB7F76F2FB5E47;
when 51 result = 0xBCC1E904BC1D2;
when 52 result = 0xC199BD85529C;
when 53 result = 0xC67F12E57D148;
when 54 result = 0xCB720DCEF9069;
when 55 result = 0xD07204A07897C;
when 56 result = 0xD5818DCFBA487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDFC97337B9B5F;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEA4AFA2A4900A;
when 61 result = 0xEF81BE615A27;
when 62 result = 0xF50765B6E4540;
when 63 result = 0xFA7C1819E9008;

return result<N-1:0>;

Library pseudocode for aarch64/functions/sve/FPLogB

```cpp
// FPLogB()
// ========

bits(N) FPLogB(bits(N) op, FPCRTyp 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(FPExc_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;
    return result<N-1:0>;
```

Shared Pseudocode Functions Page 3076
Library pseudocode for aarch64/functions/sve/FPMinNormal

```plaintext
// 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

```plaintext
// 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

```plaintext
// 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

```plaintext
// FPProcess()
// ===========

bits(N) FPProcess(bits(N) input)
  bits(N) result;
  assert N IN {16,32,64};
  (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 = FPInfinity(sign);
  elsif fptype == FPType_Zero then
    result = FPZero(sign);
  else
    result = FPRound(value, FPCR);
  return result;
```
Library pseudocode for aarch64/functions/sve/FPScale

// 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);
    return result;

Library pseudocode for aarch64/functions/sve/FPTrigMAdd

// 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;
        op2<N-1> = '0';
    coeff = FPTrigMAddCoefficient[x];
    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 = 0xbb60705;
                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 = 0x3f811111110f30c;
                when  3 result = 0xbf2a0a019b92fc6;
                when  4 result = 0x3ec71de351f3d22b;
                when  5 result = 0xbe5ae5e2b60f7b91;
                when  6 result = 0x3de5d8408868552f;
                when  7 result = 0x0000000000000000;
                when  8 result = 0x3ff0000000000000;
                when  9 result = 0x0be0000000000000;
                when 10 result = 0x3fa555555555536;
                when 11 result = 0xbf56c16c16c13a0b;
                when 12 result = 0x3efa0a019b1e8d8;
                when 13 result = 0xbe927e4f7282f468;
                when 14 result = 0x3e21ee96d2641b13;
                when 15 result = 0xbd8f76380fb401;
        return result<N-1:0>;
Library pseudocode for aarch64/functions/sve/FPTrigSMul

// 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>;
return result;

Library pseudocode for aarch64/functions/sve/FPTrigSSel

// 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

// 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

// 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

// HaveSVE()
//===

boolean HaveSVE()
  return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";
// HaveSVE2()
// ===========
// Returns TRUE if the SVE2 extension is implemented, FALSE otherwise.
boolean HaveSVE2()
return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE2 extension";

// HaveSVE2AES()
// =============
// Returns TRUE if the SVE2 AES extension is implemented, FALSE otherwise.
boolean HaveSVE2AES()
return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 AES extension";

// HaveSVE2BitPerm()
// =================
// Returns TRUE if the SVE2 Bit Permissions extension is implemented, FALSE otherwise.
boolean HaveSVE2BitPerm()
return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 BitPerm extension";

// 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";

// HaveSVE2SHA3()
// ==============
// Returns TRUE if the SVE2 SHA3 extension is implemented, FALSE otherwise.
boolean HaveSVE2SHA3()
return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA3 extension";

// HaveSVE2SM4()
// =============
// Returns TRUE if the SVE2 SM4 extension is implemented, FALSE otherwise.
boolean HaveSVE2SM4()
return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SM4 extension";

// 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()
// =============

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()
// ==============

boolean IsSVEEnabled(bits(2) el)
if ELUsingAArch32(el) then
    return FALSE;

// Check if access disabled in CPACR_EL1
if el IN {EL0, EL1} then
    // Check SVE at EL0/EL1
    case CPACR[].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
        if CPTR_EL2.ZEN == 'x0' 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;

constant integer MAX_PL = 256;
constant integer MAX_VL = 2048;
Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

// 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);
    else
        lower_enabled = IsFPEnabled(target_el - 1);

    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(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() 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 Constr ainUnpredictableBool(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
    transactional = TSTATE.depth > 0;
    accdesc = CreateAccessDescriptor(acctype, transactional);
    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];

    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 Constr ainUnpredictableBool(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

// 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

// 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_el2 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

enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
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;

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);
elif HaveEL(EL3) then
  vl = Min(vl, UInt(ZCR_EL3.LEN));
v = (vl + 1) * 128;
vl = ImplementedSVEVectorLength(v);
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(32) r;
    if IsInHost() then
        r = CNTHCTL_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
// =============================

// CPACR[] - non-assignment form
// =============================

CPACRTypex CPACR[]
    bits(32) r;
    if IsInHost() then
        r = CPTR_EL2;
        return r;
    r = CPACR_EL1;
    return r;

Library pseudocode for aarch64/functions/sysregisters/CPACRTypex

type CPACRTypex;
// 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;
// ESR[] - non-assignment form
// ============================

ESRType ESR[bits(2) regime] = ESRType value
bits(32) 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[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
        when EL1 ESR_EL1 = r;
        when EL2 ESR_EL2 = r;
        when EL3 ESR_EL3 = r;
        otherwise unreachable();
    return;

// ESR[] - non-assignment form
// ============================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;

// ESR[] - assignment form
// =======================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;

type 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[] - assignment form
// ================
bits(64) FAR[] = 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;

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[] - assignment form
// ================
MAIRType MAIR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;

Library pseudocode for aarch64/functions/sysregisters/MAIRType

type MAIRType;
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()
   // ------------------------------------------
   if SCR_EL3.ATA == '0' && PSTATE.EL IN {EL0, EL1, EL2} then
      return FALSE;
   elsif HCR_EL2.ATA == '0' && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
      return FALSE;
   elsif SCTLR_EL3.ATA == '0' && PSTATE.EL == EL3 then
      return FALSE;
   elsif SCTLR_EL2.ATA == '0' && PSTATE.EL == EL2 then
      return FALSE;
   elsif SCTLR_EL1.ATA == '0' && PSTATE.EL == EL1 then
      return FALSE;
   elsif SCTLR_EL2.ATA == '0' && PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
      return FALSE;
   elsif SCTLR_EL1.ATA == '0' && PSTATE.EL == EL0 && !EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
      return FALSE;
   else
      return TRUE;

Shared Pseudocode Functions
Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

// AArch64.CheckSystemAccess()
// --------------------------
// Checks if an AArch64 MSR, MRS or SYS instruction is allowed from the current exception level and security state.
// Also checks for traps by TIDCP and NV access.
AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bits(5) rt, bit read)

boolean unallocated = FALSE;
boolean need_secure = FALSE;
bits(2) min_EL;

if TSTATE.depth > 0 && CheckTransactionalSystemAccess(op0, op1, crn, crm, op2, read) then
    FailTransaction(TMFailure_ERR, FALSE);

// Check for traps by HCR_EL2.TIDCP
if PSTATE.EL IN {EL0, EL1} && HCR_EL2.TIDCP == '1' && Oper == 'x1' && crn == '1x1' then
    FailTransaction(TMFailure_ERR, FALSE);

// At EL0, it is IMPLEMENTATION_DEFINED whether attempts to execute system
// register instructions with reserved encodings are trapped to EL2 or UNDEFINED
rcs_el0_trap = boolean IMPLEMENTATION_DEFINED "Reserved Control Space EL0 Trapped";
if PSTATE.EL == EL1 || rcs_el0_trap then
    AArch64.SystemAccessTrap(EL2, 0x18); // Exception_System Register Trap

// Check for unallocated encodings
case op1 of
    when '00x', '010'
        min_EL = EL1;
    when '011'
        min_EL = EL0;
    when '100'
        min_EL = EL2;
    when '101'
        if !HaveVirtHostExt() then UNDEFINED;
        min_EL = EL2;
    when '110'
        min_EL = EL3;
    when '111'
        min_EL = EL1;
        need_secure = TRUE;

if UInt(PSTATE.EL) < UInt(min_EL) then
    // Check for traps on read/write access to registers named _EL2, _EL02, _EL12 from EL1 when
    // EL2 is enabled in the current security state, and the HCR_EL2.NV bit is set
    nv_access = HaveNVExt() && min_EL == EL2 && PSTATE.EL == EL1 && HCR_EL2.NV == '1';
    if !nv_access then
        UNDEFINED;
    elsif need_secure && !IsSecure() then
        UNDEFINED;

    // Check for traps on read/write access to registers named _EL2, _EL02, _EL12 from EL1 when
    // EL2 is enabled in the current security state, and the HCR_EL2.NV bit is set
    // if !nv_access then
    // UNDEFINED;
    // elsif need_secure && !IsSecure() then
    // UNDEFINED;

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> == '11010110100111110000';

Library pseudocode for aarch64/functions/system/AArch64.NextRandomTagBit

// 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

// 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

// 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

// 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 crm, integer op2);
Library pseudocode for aarch64/functions/system/AArch64.SysRegRead

// 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

// 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

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;

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;

Library pseudocode for aarch64/functions/system/BTypeNext

bits(2) BTypeNext;

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

```c
// SetBTypeNext()
// ============
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;
```

Library pseudocode for aarch64/functions/system/_ChooseRandomNonExcludedTag

```c
// 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 expected 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().

bits(4) _ChooseRandomNonExcludedTag(bits(16) exclude);
```

Library pseudocode for aarch64/functions/tme/CheckTMEEnabled

```c
// CheckTMEEnabled()
// ===========
// Returns TRUE if access to TME instruction is enabled, FALSE otherwise.

CheckTMEEnabled()
    if PSTATE.EL IN {EL0, EL1, EL2} && HaveEL(EL3)
        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

```c
// 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 0000 0000 000' return TRUE;      // MSR: ICC_PPR_SET_EL1
        when '0 11 000 1011 1100 111' return TRUE;      // MRS: ICC_HPI𝕏_EL1
        when 'x 11 xxx 1110 xxxx xxx' return TRUE;      // MRS: ICC_HPM_EL1
        otherwise return FALSE;
```

Library pseudocode for aarch64/functions/tme/CommitTransactionalWrites

```c
// 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();
    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_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;
    [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>;
        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(32) FPCR, // Floating-point Control Register bits(32) FPSR, // Floating-point Status Register bits(32) 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

```c
TMState TSTATE;
```

Library pseudocode for aarch64/functions/tme/TakeTransactionCheckpoint

// TakeTransactionCheckpoint()
// ===========================
// Captures part of the PE registers into the transaction checkpoint.

```c
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];
        end
    TSTATE.FPCR = FPCR;
    TSTATE.FPSR = FPSR;
    return;
```

Library pseudocode for aarch64/functions/tme/TransactionStartTrap

// TransactionStartTrap()
// ======================
// Traps the execution of TSTART instruction.

```c
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(32) spsr)

    if TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    SynchronizeContext();

    sync_errors = HaveIESB() &amp; SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' &amp;&amp; SCR_EL3.NMEA == '1' &amp;&amp; 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
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' &amp;&amp; spsr<4> == '1' &amp;&amp; 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
            end if

        else                                     // Return to AArch64
            // ELR_ELx[63:56] might include a tag
            new_pc = AArch64.BranchAddr(new_pc);

        if UsingAAarch32() 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

```
```

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;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);
    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;

mask = 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 Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));

// optimization of first step:
// wmask = Replicate(wmask_or<0> : '0', 32);
wmask = ((wmask
AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
wmask = ((wmask
AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

if diff<6> != '0' then // borrow from S - R
  wmask = wmask AND tmask;
else
  wmask = wmask OR tmask;

return (wmask<M-1:0>, tmask<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 = UInt(imms);
  integer R = 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));

  // borrowed from S - R
  return FALSE;
Library pseudocode for aarch64/instrs/integer/shiftreg/DecodeShift

// 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;

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftReg

// 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;

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftType

enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};

Library pseudocode for aarch64/instrs/logicalop/LogicalOp

enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

Library pseudocode for aarch64/instrs/memory/memop/MemAtomicOp


Library pseudocode for aarch64/instrs/memory/memop/MemOp

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
// Prefetch()
// =========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(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;

Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

enumeration MemBarrierOp   {
   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
};

Library pseudocode for aarch64/instrs/system/hints/syshintop/SystemHintOp

enumeration SystemHintOp {
   SystemHintOp_NOP,
   SystemHintOp_YIELD,
   SystemHintOp_WFE,
   SystemHintOp_WFI,
   SystemHintOp_SEV,
   SystemHintOp_SEVL,
   SystemHintOp_DGH,
   SystemHintOp_ESB,
   SystemHintOp_PSB,
   SystemHintOp_TSB,
   SystemHintOp_BTI,
   SystemHintOp_CSDB
};

Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
   PSTATEField_PAN, // Armv8.1
   PSTATEField_UAO, // Armv8.2
   PSTATEField_DIT, // Armv8.4
   PSTATEField_SSBBS,
   PSTATEField_TCO, // Armv8.5
   PSTATEField_SP
};
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; // S1E01R
    when '100 0111 1000 101' return Sys_AT; // S1E01W
    when '000 0111 1010 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 0100 001' return Sys_DC; // CVAC
    when '000 0111 1010 010' return Sys_DC; // CSW
    when '011 0111 1010 001' return Sys_DC; // CVAU
    when '011 0111 1110 010' return Sys_DC; // CIVAC
    when '000 0111 1110 010' return Sys_DC; // CISW
    when '011 0111 1101 001' return Sys_DC; // CVADP
    when '000 0111 0001 000' return Sys_IC; // IALLUIS
    when '000 0111 0101 000' return Sys_IC; // IALLU
    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; // VALE1
    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; // ALLE2
    when '000 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; // 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};
VBitOp enumeration

VBitOp_VBIF,
VBitOp_VBIT,
VBitOp_VBSL,
VBitOp_VEOR;

CompareOp enumeration

CompareOp_GT,
CompareOp_GE,
CompareOp_EQ,
CompareOp_LE,
CompareOp_LT;

ImmediateOp enumeration

ImmediateOp_MOVI,
ImmediateOp_MVNI,
ImmediateOp_ORR,
ImmediateOp_BIC;

Reduce function

```c
bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
  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);
  lo = Reduce(op, input<half-1:0>, esize);
  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);
    when ReduceOp_FMAX
      result = FPMax(lo, hi, FPCR);
    when ReduceOp_FADD
      result = FPAdd(lo, hi, FPCR);
    when ReduceOp_ADD
      result = lo + hi;
  return result;
```

ReduceOp enumeration

ReduceOp_FMINNUM,
ReduceOp_FMAXNUM,
ReduceOp_FMIN,
ReduceOp_FMAX,
ReduceOp_FADD,
ReduceOp_ADD;
AddressDescriptor AArch64.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    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 &&
            s2desc.memattrs.inner.attrs != '10') ||
           (apply_force_writeback && s1desc.memattrs.memtype == MemType_Device)
        then
            result.memattrs.memtype = MemType_Device;
            if s2desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s2desc.memattrs.device;
            elseif s2desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s1desc.memattrs.device;
            else
                // Both Device
                result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                        s2desc.memattrs.device);
            end
            result.memattrs.tagged = FALSE;
        else
            result.memattrs.memtype = MemType_Normal;
            result.memattrs.device = DeviceType UNKNOWN;
            if apply_force_writeback then
                if s2desc.memattrs.memtype == MemType_Normal &&
                   s2desc.memattrs.inner.attrs == '10' then
                    result.memattrs.inner.attrs = MemAttr_WB; // force Write-back
                elseif s2desc.memattrs.inner.attrs == '11' then
                    result.memattrs.inner.attrs = MemAttr_NC;
                end
                result.memattrs.outer = result.memattrs.inner;
            else
                result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner,
                                                             s2desc.memattrs.inner);
                result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer,
                                                             s2desc.memattrs.outer);
            end
            result.memattrs.shareable = (s1desc.memattrs.shareable ||
                                         s2desc.memattrs.shareable);
            result.memattrsoutershareable = (s1desc.memattrsoutershareable ||
                                             s2desc.memattrsoutershareable);
            result.memattrs.tagged = MemAttrDefaults(result.memattrs);
        end
    end

    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;
// AArch64.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

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 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
        end case;
    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 == '11100000' then // Normal, Tagged if 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 = (memattrs.inner.attrs == MemAttr WB &&
                          memattrs.inner.hints == MemHint RWA &&
                          memattrs.outer.attrs == MemAttr WB &&
                          memattrs.outer.hints == MemHint RWA);
    else
        Unreachable();                          // Reserved, handled above
    end if;

return MemAttrDefaults(memattrs);
Libary pseudocode for aarch64/translation/attrs/AArch64.TranslateAddressS1Off

// 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;
Top = AddrTop(vaddress, (acctype == AccType_IFETCH), PSTATE.E); 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.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_ELx.I
  if 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;
  endif
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 IssSecure() then '0' else '1';
result.addrdesc.fault = AArch64.NoFault();
return result;
Library pseudocode for aarch64/translation/checks/AArch64.AccessIsPrivileged

// AArch64.AccessIsPrivileged()
// ============================

boolean AArch64.AccessIsPrivileged(AccType acctype)

    el = AArch64.AccessUsesEL(acctype);
    if el == EL0 then
        ispriv = FALSE;
    elsif el == EL3 then
        ispriv = TRUE;
    elsif el == EL2 && (!IsInHost() || HCR_EL2.TGE == '0') then
        ispriv = TRUE;
    elsif HaveUAOExt() && PSTATE.UAO == '1' then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);
    return ispriv;

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;
    else
        return PSTATE.EL;
Library pseudocode for aarch64/translation/checks/AArch64.CheckPermission
// AArch64.CheckPermission()
// -----------------------
// Function used for permission checking from AArch64 stage 1 translations

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.AccessIsPrivileged(acctype);

pan = if HavePANExt() then PSTATE.PAN else '0';

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 && ispriv && (is_ldst || is_ats1xp) then
    priv_r = FALSE;
    priv_w = FALSE;

user_xn = perms.xn == '1' || (user_w && wxn);

priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

if ispriv then
    (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 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;
Library pseudocode for aarch64/translation/checks/AArch64.CheckS2Permission

// AArch64.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch64 stage 2 translations

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;
  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 faults 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)
    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;
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());

match = FALSE;
ispriv = AArch64.AccessIsPrivileged(acctype);

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;
        Halt(reason);
    else
        // Fault will be reported and cancelled
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
else
    return AArch64.DebugFault(acctype, iswrite);

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);
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.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);
}

Library pseudocode for aarch64/translation/faults/AArch64.PermissionFault

// AArch64.PermissionFault()
// =========================

FaultRecord AArch64.PermissionFault(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_Permission, ipaddress, NS, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
}

Library pseudocode for aarch64/translation/faults/AArch64.TranslationFault

// AArch64.TranslationFault()
// ==========================

FaultRecord AArch64.TranslationFault(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_Translation, ipaddress, NS, level, acctype, iswrite, extflag, errortype, secondstage, s2fs1walk);
}
// AArch64.CheckAndUpdateDescriptor()
// ==================================
// Check and update translation table descriptor if hardware update is configured

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 || ConstrainUnpredictable(Unpredictable_AFUPDATE) == Constraint_TRUE
    hw_update_AF = TRUE;
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;
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;
accdesc = CreateAccessDescriptor(AccType_ATOMICRW, FALSE);
desc = _Mem[descaddr2, 8, accdesc];
case el of
  when EL3
    reversedescriptors = SCTLR_EL3.EE == '1';
  when EL2
    reversedescriptors = SCTLR_EL2.EE == '1';
  otherwise
    reversedescriptors = SCTLR_EL1.EE == '1';
if reversedescriptors then
  desc = BigEndianReverse(desc);
if hw_update_AF then
  desc<10> = '1';
if hw_update_AP then
  desc<7> = (if secondstage then '1' else '0');
_Mem[descaddr2,8,accdesc] = if reversedescriptors then BigEndianReverse(desc) else desc;
return fault;
// 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';  
elsif HasS2Translation() then  
    s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';  
else  
    s1_enabled = SCTLR[].M == '1';  

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;  
    
    if !IsFault(S1.addrdesc) && UsingAArch32() && HaveTrapLoadStoreMultipleDeviceExt() && AArch32.ExecutingLSMInstr() then  
        nTLSMD = if S1TranslationRegime() == EL2 then SCTLR_EL2.nTLSMD else SCTLR_EL1.nTLSMD;  
        if nTLSMD == '0' then  
            S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);  
        // Check for unaligned data accesses to Device memory  
        if (!wasaligned && acctype != AccType_DCZVA) || (!wasaligned && acctype != AccType_IFETCH) then  
            S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);  
        // 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) && !AArch64.CheckPermission(S1.addrdesc, vaddress, S1.level,  
            S1.addrdesc.paddress.NS,  
            acctype, iswrite) then  
            S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,  
                acctype, iswrite,  
                secondstage, s2fs1walk);  
        // 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))
        && 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, size);

    // 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);

    // Check for protected table walk
    if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
        S2.addrdesc.memattrs.memtype == MemType_Device) then
        S2.addrdesc.fault = AArch64.PermissionFault(ipaddress, NS, S2.level, acctype, 
            iswrite, secondstage, s2fs1walk);

    // Check and update translation table descriptor if required
    S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault, 
        secondstage, vaddress, acctype, 
            iswrite, s2fs1walk, hwupdatewalk);

result = AArch64.CombineS1S2Desc(S1, S2.addrdesc);
else
    result = S1;

return result;
// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType accype, boolean iswrite, integer size, boolean hwupdatewalk)

    assert HasS2Translation();
    s2fs1walk = TRUE;
    wasaligned = TRUE;
    return AArch64.SecondStageTranslate(S1, vaddress, accype, iswrite, wasaligned, s2fs1walk, size, hwupdatewalk);

// AArch64.TranslateAddress()  
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType accype, boolean iswrite, boolean wasaligned, integer size)

    result = AArch64.FullTranslate(vaddress, accype, iswrite, wasaligned, size);
    if !(accype IN {AccType PTW, AccType IC, AccType AT}) && !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, accype, 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 acctype, boolean iswrite, boolean secondstage,
boolean s2fs1walk, integer size)

    if !secondstage then
        assert !ELUsingAArch32(S1TranslationRegime());
    else
        assert IsSecureEL2Enabled() || ( HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) ) && HasS2TranslationRegime();

    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

    descaddr.memattrs.memtype = MemType_Normal;

    // Derived parameters for the page 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(accctype);
        top = AddrTop(inputaddr, (accctype == 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;
            inputsize_min = 64 - (if !HaveSmallPageTblExt() 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 = TCR_EL3.PS;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top>);
                baseregister = TTBR0_EL3;
                descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
                lookupsecure = SCTLR_EL3.EE == '1';
                singlepriv = TRUE;
                disable = FALSE;
                update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.MA == '1';
                update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
                hierattrssubstitution = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
            else if ELIsInHost(el) then
                if inputaddr<top> == '0' then
                    largegrain = TCR_EL2.TG0 == '01';
                    midgrain = TCR_EL2.TG0 == '10';
                    inputsize = 64 - UInt(TCR_EL2.T0SZ);
                    inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                    inputsize_min = 64 - (if !HaveSmallPageTblExt() 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 = TCR_EL3.PS;
                        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top>);
                        baseregister = TTBR0_EL3;
                        descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
                        lookupsecure = SCTLR_EL3.EE == '1';
                        singlepriv = TRUE;
                        disable = FALSE;
                        update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.MA == '1';
                        update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
                        hierattrssubstitution = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
                    else
                        Shared Pseudocode Functions
                        Page 3131
basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top:inputsize>

disabled = TCR_EL2.EPD0 == '1' || (PSTATE.EL == EL0 & HaveE0PDExt()) & TCR_EL2.E0PD0 == '01';
baseregister = TTBR0_EL2;
descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
hierattributesdisabled = AArch64.HaveHPDExt() & & TCR_EL2.HPD0 == '1';
else
    inputsize = 64 - UInt(TCR_EL2.T1SZ);
largegrain = TCR_EL2.TG1 == '11'; // TG1 and TG0 encodings differ
midgrain = TCR_EL2.TG1 == '01';
inputsize_max = if Have52BitVAExt() & & largegrain then 52 else 48;
inputsize_min = 64 - (if HaveSmallPageTblExt() 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 = TCR_EL2.IPS;
reversedescriptors = SCTLR_EL2.EE == '1';
lookupsecure = if IsSecureEL2Enabled() then IsSecure() else FALSE;
singlevip = FALSE;
update_AF = HaveAccessFlagUpdateExt() & & TCR_EL2.HA == '1';
update_AP = HaveDirtyBitModifierExt() & & update_AF & & TCR_EL2.HD == '1';
elsif el == EL2 then
    inputsize = 64 - UInt(TCR_EL2.T0SZ);
largegrain = TCR_EL2.TG0 == '01';
midgrain = TCR_EL2.TG0 == '10';
inputsize_max = if Have52BitVAExt() & & largegrain then 52 else 48;
inputsize_min = 64 - (if HaveSmallPageTblExt() 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 = TCR_EL2.PS;
basefound = inputsize >= inputsize_min & & inputsize <= inputsize_max & & IsZero(inputaddr<top:inputsize>

disabled = TCR_EL1.EPD0 == '1' || (PSTATE.EL == EL0 & & HaveE0PDExt()) & & TCR_EL1.E0PD0 == '01';
baseregister = TTBR0_EL1;
descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0, secondstage);
hierattributesdisabled = AArch64.HaveHPDExt() & & TCR_EL1.HPD0 == '1';
else
    inputsize = 64 - UInt(TCR_EL1.T1SZ);
largegrain = TCR_EL1.TG1 == '11'; // TG1 and TG0 encodings differ
midgrain = TCR_EL1.TG1 == '01';
inputsize_max = if Have52BitVAExt() & & largegrain then 52 else 48;
inputsize_min = 64 - (if HaveSmallPageTblExt() then 39 else if largegrain then 47 else 48);

if inputsize < inputsize_min then
    c = ConstrantUnpredictable(Unpredictable_RESTnSZ);
    assert c IN {Constraint_FORCE, Constraint_FAULT};
    if c == Constraint_FORCE then inputsize = inputsize_min;
    basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputaddr);
    disabled = TCR_EL1.EPD1 == '1' || (PSTATE_EL == EL0 && HaveE0PDExt() && TCR_EL1.E0PD1 == '1');
    baseregister = TTBR1_EL1;
    descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1, secondstage);
    hierattrdisabled = 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)
        // and 512MB (2^29 bytes) otherwise
    else if midgrain then
        grainsize = 14;                                             // Log2(16KB page size)
        firstblocklevel = 2;
    else // Small grain
        grainsize = 12;                                             // Log2(4KB page size)
        firstblocklevel = 1;
        stride = grainsize - 3;                                         // Log2(page size / 8 bytes)
        level = 4 - (1 + (inputsize - grainsize - 1) DIV stride);
        // 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;
        endif
        // 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';
        else if !s1_nonsecure then
            // When walk is Secure and in the Non-secure IPA space,
            // access is specified by VSTCR_EL2.SA
            nsaccess = VSTCR_EL2.SA;
        else
            // When walk is Secure and in the Non-secure IPA space,
            // if VSTCR_EL2.SA specifies the Non-secure PA space, access is specified by VTCR_EL2.NSA
            nsaccess = VTCR_EL2.NSA;
        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;
        endif
        inputsize  = 64 -UInt(t0size);
        largegrain = tg0 == '01';
        midgrain   = tg0 == '10';
inputsize_max = if Have52BitPAExt() & PAMax() == 52 & largegrain then 52 else 48;
inputsize_min = 64 - (if !HaveSmallPageTblExt() 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;
   level = 3 - startlevel;
   firstblocklevel = (if Have52BitPAExt() then 1 else 2);  // Largest block is 4TB (2^42 bytes) and 512MB (2^29 bytes) otherwise
elsif midgrain then
   grainsize = 14;
   level = 3 - startlevel;
   firstblocklevel = 2;                                    // Largest block is 32MB (2^25 bytes)
else // Small grain
   grainsize = 12;
   if HaveSmallPageTblExt() & startlevel == 3 then
      level = startlevel;
   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();
      when Constraint_FORCEONLYCHECK
         // As FORCE, except use the configured inputsize in the size checks below
         inputsize = PAMax();
      when Constraint_FAULT
         // Generate a translation fault
Shared Pseudocode Functions
basefound = FALSE;
otherwise
    
Unreachable();

// 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(Number 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(Number of entries*8)
if outputsize == 52 then
    z = (if baselowerbound < 6 then 6 else baselowerbound);
    baseaddress = 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' && SITranslationRegime() &&
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 = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
desc = _Mem[descaddr2, 8, accdesc];
if reversedescriptors then desc = BigEndianReverse(desc);
if desc<0> == '0' || (desc<1:0> == '01' && (level == 3 || 
    (HaveBlockBBM() && IsBlockDescriptorNTBitValid() && desc<16> == '1'))) then
    // 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     // Block(01) or Page(11)
    blocktranslate = TRUE;
else                                                    // Table(11)
    if (outputsize < 52 && largegrain && !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;
elif midgrain then
    if level == 3 then
        num_ch_entries = 7;
    else num_ch_entries = 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: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,s1.nonsecure, level, acctype, iswrite, secondstage, s2fs1walk);
      return result;
    else
      result.descupdate.AF = TRUE;
  end if

  if update_AP && desc<51> == '1' then
    // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
    if !secondstage && desc<7> == '1' then
      desc<7> = '0';
      result.descupdate.AP = TRUE;
    elsif secondstage && desc<7> == '0' then
      desc<7> = '1';
      result.descupdate.AP = TRUE;
    end if

  end if

  // Required descriptor if AF or AP[2]/S2AP[2] needs update
  result.descupdate.descaddr = descaddr;

  if apply_nvnl_effect then
    pxn = desc<54>;                                       // Bit[54] of the block/page descriptor holds PXN instead of UXN
    xn = '0';                                             // XN is '0'
    ap = desc<7>:'01';                                    // Bit[6] of the block/page descriptor is the AP[2]
  else
    xn = desc<54>;                                        // Bit[54] of the block/page descriptor holds UXN
    pxn = desc<53>;                                       // Bit[53] of the block/page descriptor holds PXN
  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;                       // Domains not used
  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
    result.perms.pxn     = pxn OR pxn_table;
    // PXN, nG and AP[1] apply in EL1&0 or EL2&0 stage 1 translation regimes
    if singlepriv then
      result.perms.ap<1>  = ap<1> AND NOT(ap_table<0>);   // Force privileged only
      if IsSecure() then
        result.nG = nG OR ns_table;
      else
        Shared Pseudocode Functions
      end if
    end if
  end if
result.nG = nG;
else
    result.perms.ap<1> = '1';
    result.perms.pxn = '0';
    result.nG = '0'; // Stage 1 block or pages might be guarded
    result.GP = desc<50>;
    result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
    result.addrdesc.paddress NS = memattr<3> OR ns_table;
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>;
    result.perms.pxn = '0';
    result.nG = '0';
    if s2fslwalk then
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_PTW);
    else
        result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
    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
    EDSCR.ITO = '0'; // Clear ITR overrun flag

// If halted and the ITR is not empty then it is UNPREDICTABLE whether the ESCR.ERR is cleared.
// The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
// in the pseudocode.
if Halted() && ESCR.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

```plaintext
// DebugTargetFrom()
// =============

bits(2) DebugTargetFrom(boolean secure)
  if HaveEL(EL2) && !secure then
    if ELUsingAAArch32(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) && HighestELUsingAAArch32() && secure then
    target = EL3;
  else
    target = EL1;
  return target;
```

Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```plaintext
// 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 ELUsingAAArch32(EL1) then
    return DBG0SDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
  else
    return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```
Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
// The access may also be subject to OS Lock, power-down, etc.
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)
// The access may also be subject to OS Lock, power-down, etc.
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';
    else
      return !IsSecure();
  else
    return FALSE;
// AllowExternalTraceAccess()  
// ==========================  
// Returns TRUE if an external Trace access to the Trace registers is allowed, FALSE otherwise.

boolean AllowExternalTraceAccess()
  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 (ExternalNoninvasiveDebugEnabled() &&
    (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled() ||
    (ELUsingAArch32(EL1) && PSTATE.EL == EL0 && SDER.SUNIDEN == '1')));
Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

// ExternalNoninvasiveDebugEnabled()
// =================================
// This function returns TRUE if the ARMv8.4-Debug is implemented, otherwise this
// function is IMPLEMENTATION DEFINED.
// In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEND
// 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, this function returns the state of the (DBGEN AND SPIDEN) signal.
// 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 ARMv8.4-Debug
// 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();
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccType acctype, integer size)
  assert !ELUsingAArch32(S1TranslationRegime());
  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
    // CONSTRAINED 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 CONSTRAINED
    // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
    // at the address DBGVR_EL1[n]+2.
    if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

  match = value_match && state_match && enabled;

return match;
Library pseudocode for shared/debug/breakpoint/AArch64.BreakpointValueMatch
// 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()) ||              // Context matching
          dbgtype == '010x' ||                                             // Reserved
          (dbgtype != '0x0x' && !context_aware) ||                         // Context matching
          (dbgtype == '1xxx' && !HaveEL(EL2))) then                        // EL2 extension
        (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_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                     // "vaddress" is word aligned
            byte_select_match = TRUE;
        top = AddrTop(vaddress, TRUE, PSTATE.EL);
        BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
    elsif match_cid then
        if IsInHost() then
            BVR_match = (CONTEXTIDR_EL2 == DBGBVR_EL1[n]<31:0>);
        else
            BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
        end
    elseif match_cid1 then
        BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1 == DBGBVR_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(DBGBVR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGBVR_EL1[n]<47:32>;
    BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                          vmid == bvr_vmid);
elsif match_cid2 then
    BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                  DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2);

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;

// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

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 == Constraint_DISABLED then return FALSE;
  // Otherwise the HMC,SSC,PxC values are either valid or the values returned by
  // CheckValidStateMatch are valid.

  EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
  EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
  EL1_match = PxC<0> == '1';
  EL0_match = PxC<1> == '1';

  if HaveNV2Ext() && acctype == AccType_NV2REGISTER && !isbreakpnt then
    priv_match = EL2_match;
  elsif !ispriv && !isbreakpnt then
    priv_match = EL0_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;
    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.
      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 {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
      end case

      if linked then
        vaddress = bits(64) UNKNOWN;
        linked_to = TRUE;
        linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);
      end if

      return priv_match && security_state_match && (!linked || linked_match);
    end if
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;

// EL3 using AArch64 only
if (!HaveEL(EL3) || HighestELUsingAArch32()) && HMC:SSC:PxC == '11000' 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 {'01100','10100','x11x1'} then
reserved = TRUE;

// Values that are not allocated in any architecture version
if (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

```plaintext
enumeration CrossTriggerOut  {
    CrossTriggerOut_DebugRequest,
    CrossTriggerOut_IRQ,
    CrossTriggerOut_TraceExtIn0,
    CrossTriggerOut_TraceExtIn1,
    CrossTriggerOut_TraceExtIn2,
    CrossTriggerOut_TraceExtIn3;
}

enumetration CrossTriggerIn  {
    CrossTriggerIn_CrossHalt,
    CrossTriggerIn_PMUOverflow,
    CrossTriggerIn_TraceExtOut0,
    CrossTriggerIn_TraceExtOut1,
    CrossTriggerIn_TraceExtOut2,
    CrossTriggerIn_TraceExtOut3;
}
```

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```plaintext
// 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  // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED " signal slave-generated error";
    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 EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
    EDSCR.RXO = '1';  EDSCR.ERR = '1';  // Overrun condition: ignore write
    return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if !UsingAArch32() then
        ExecuteA64(0x0D330501<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;
    // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.RXfull = bit UNKNOWN;
        DBGDTRRX_EL0 = bits(32) 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;

// DBGDTRRX_EL0[] (external read)
// =================================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
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 "signal slave-generated error";
    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

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(32) 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"
                // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
                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;
else
    // 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<31:0>;
    DTRTX = value<63:32>;        // 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;
Library pseudocode for shared/debug/dccanditr/EDITR

// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "signal slave-generated error";
        return;
    if EDSCR.ERR == '1' then return;                    // Error flag set: 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;
Library pseudocode for shared/debug/halting/DCPSInstruction
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
      end
    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(32) UNKNOWN;
          ESR[] = bits(32) UNKNOWN;
          DLR_EL0 = bits(64) UNKNOWN;
          DSPSR_EL0 = bits(32) UNKNOWN;
          if HaveIAOExt() then PSTATE.UAO = '0';
          if HaveMTEExt() then PSTATE.TCO = '1';
          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.
            if !ConstrainUnpredictableBool(UnpredictableIESBinDebug) then
            end
          end
        end
      end
    end
  end
end
sync_errors = FALSE;
if sync_errors then
  SynchronizeErrors();
return;

Library pseudocode for shared/debug/halting/DRPSInstruction

// DRPSInstruction()
// ================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state
DRPSInstruction()

  SynchronizeContext();
  sync_errors = HaveIESB() &amp; SCTLR[].IESB == '1';
  if HaveDoubleFaultExt() &amp; !UsingAArch32() then
    sync_errors = sync_errors || (SCR_EL3.EA == '1' &amp;&amp; SCR_EL3.NMEA == '1' &amp;&amp; PSTATE.EL == EL3);
  if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
    sync_errors = FALSE;
  if sync_errors then
    SynchronizeErrors();
  SetPSTATEFromPSR(PSR[]);

  // PSTATE.{N,Z,C,V,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
  // behave as if UNKNOWN.
  if UsingAArch32() then
    // In AArch32, all instructions are T32 and unconditional.
    PSTATE.T = '00000000';  // PSTATE.J is RES0
    PSTATE.IT = '00000000';
    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(32) UNKNOWN;
    UpdateEDSCRFields();
  return;

Library pseudocode for shared/debug/halting/DebugHalt

constant bits(6) DebugHalt_Breakpoint = '000111';
constant bits(6) DebugHalt_EDBGRQ = '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(32) spsr;
if UsingAArch32() then
    new_pc = ZeroExtend(DLR);
    spsr = DSPSR;
else
    new_pc = DLR_EL0;
    spsr = DSPSR_EL0;
// If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
SetPSTATEFromPSR(spsr);                   // Can update privileged bits, even at EL0
if UsingAArch32() then
    if ConstrainsUnpredictableBool(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' && ConstrainsUnpredictableBool(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

```plaintext
// Halt()
// =====

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr();
    spsr = GetPSRFromPSTATE();

    if UsingAArch32() then
        // If entering from AArch32 state, spsr<21> is the DIT bit which has to be moved for DSPSR
        spsr<24> = spsr<21>;
        spsr<21> = PSTATE.SS;  // Always save the SS bit
    else
        DLR = preferred_restart_address<31:0>
        DSPSR = spsr;

    EDSCR.ITE = '1';
    EDSCR.IE0 = '0';
    if IsSecure() then
        EDSCR.SDD = '0';  // If entered in Secure state, allow debug
    else if HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        assert EDSCR.SDD == '1';  // Otherwise EDSCR.SDD is RES1
        EDSCR.MA = '0';

    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
        // In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';
        PSTATE.I = '1';  // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
        PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;  // Signal entered Debug state
    UpdateEDSCRFields();  // Update EDSCR 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() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```
// Halted()
// =========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                     // Halted

// 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();
    
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                  // Restarting

// StopInstructionPrefetchAndEnableITR
StopInstructionPrefetchAndEnableITR();
Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```plaintext
// 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

```plaintext
// 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);
  elsif HaltingStep_SteppedEX() then
    Halt(DebugHalt_Step_Exclusive);
  else
    Halt(DebugHalt_Step_Normal);

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 (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);

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);
Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

// CheckSoftwareAccessToDebugRegisters()
// --------------------------------------
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

// ExternalDebugRequest()
// ----------------------

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();

Library pseudocode for shared/debug/haltingevents/RunHaltingStep

// RunHaltingStep()
// ------------------

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
    boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    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
        active = 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;

Shared Pseudocode Functions
Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

// ExternalDebugInterruptsDisabled()
// ================================
// Determine whether EDSCR disables interrupts routed to 'target'

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    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

enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIRQ,
                            InterruptID_COMMRX, InterruptID_COMMTX};

Library pseudocode for shared/debug/interrupts/SetInterruptRequestLevel

// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);

Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

// CreatePCSample()
// ===============
CreatePCSample()
    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;
    pc_sample.has_el2 = EL2Enabled();
    if EL2Enabled() then
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elsif !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() & ELUsingAArch32(EL2) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
        pc_sample.el0h = PSTATE.EL == EL0 & IsInHost();
    return;
Library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo

// EDPCSRlo[] (read) // ================

bits(32) EDPCSRlo[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "signal slave-generated error";
    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() && 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.NS = pc_sample.ns;
            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 "signal slave-generated error";
  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().
if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
  if MDSCR_EL1.SS == '1' && 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(32) spsr)
    assert Halted() || Restarting() || PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        elsif UsingAArch32() then
            enabled_at_source = AArch32.GenerateDebugExceptions();
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();
        end

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr); assert valid;

            secure = IsSecureBelowEL3() || dest == EL3;
            if ELUsingAArch32(dest) then
                enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest, secure);
            else
                mask = spsr<9>
                enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);
            end

            ELd = DebugTargetFrom(secure);
            if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
                SS_bit = spsr<21>;
            end
        end
    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;

Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

// 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();
Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

// 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 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();

Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

  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';

  return syndrome;
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_CP15RRTTrap, // Trapped AArch32 MCR or MRC access to CP15
    Exception_CP14RTTrap, // Trapped AArch32 MCR or MRC access to CP14
    Exception_CP14DTTrap, // Trapped AArch32 LDC or STC access to CP14
    Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
    Exception_FPIDTrap, // Trapped BXJ instruction not supported in Armv8
    Exception_PACTrap, // Trapped invalid PAC use
    Exception_CP14RRTTrap, // Trapped MRRC access to CP14 from AArch32
    Exception_IllegalState, // Illegal Execution state
    Exception_SupervisorCall, // Supervisor Call
    Exception_HypervisorCall, // Hypervisor Call
    Exception_CP14RRTTrap, // Trapped MRRC access to CP14 from AArch32
    Exception_IllegalState, // Illegal Execution state
    Exception_SupervisorCall, // Supervisor Call
    Exception_HypervisorCall, // Hypervisor Call
    Exception_SystemRegisterTrap, // Trapped MRS or MSR system register access
    Exception_ERetTrap, // Trapped invalid ERET use
    Exception_InstructionAbort, // Instruction Abort or Prefetch Abort
    Exception_PCApignment, // PC alignment fault
    Exception_DataAbort, // Data Abort
    Exception_NV2DataAbort, // Data abort at EL1 reported as being from EL2
    Exception_PACFail, // PAC Authentication failure
    Exception_SPApignment, // SP alignment fault
    Exception_FPTrapException, // IEEE trapped FP exception
    Exception_SError, // SError interrupt
    Exception_Breakpoint, // (Hardware) Breakpoint
    Exception_SoftwareStep, // 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, // HCPTR 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

**type ExceptionRecord is**

**Exception** exceptype, // Exception class
**bits(25)** syndrome, // Syndrome record
**bits(64)** vaddress, // Virtual fault address
**boolean** ipavalid, // Physical fault address for second stage faults is valid
**bits(1)** NS, // Physical fault address for second stage faults is Non-secure or secure
**bits(52)** ipaddress) // Physical fault address for second stage faults

Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

/// 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.vaddress = **Zeros**();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = **Zeros**();

    return r;
// ReservedValue()
// ===============
ReservedValue()
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();

// UnallocatedEncoding()
// =====================
UnallocatedEncoding()
    if UsingAArch32() && AArch32.ExecutingCP10or11Instr() then
        FPEXC.DEX = '0';
    if UsingAArch32() && !AArch32.GeneralExceptionsToAArch64() then
        AArch32.TakeUndefInstrException();
    else
        AArch64.UndefinedFault();

// 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_AsyncExternal result = '011001';
        when Fault_AsyncExternalOnWalk result = '010001';
        when Fault_AsyncParity result = '100001';
        when Fault_AsyncParityOnWalk result = '100010';
        when Fault_Debug result = '100010';
        when Fault_TLBConflict result = '110000';
        when Fault_HWUpdateAccessFlag result = '110001';
        when Fault_Lockdown result = '110100'; // IMPLEMENTATION DEFINED
        when Fault_Exclusive result = '110101'; // IMPLEMENTATION DEFINED
        otherwise result = Unreachable();
    return result;
// IPAvail()  
// =========
// Return TRUE if the IPA is reported for the abort

boolean IPAvail(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;

// 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});

boolean IsAsyncAbort(FaultRecord fault)
return IsAsyncAbort(fault.statuscode);

// IsDebugException()  
// ==================

boolean IsDebugException(FaultRecord fault)
assert fault.statuscode != Fault_None;
return fault.statuscode == Fault_Debug;

// 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});

boolean IsExternalAbort(FaultRecord fault)
return IsExternalAbort(fault.statuscode);
// 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;

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
return IsExternalSyncAbort(fault.statuscode);

// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
return addrdesc.fault.statuscode != Fault_None;

// 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);

// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
assert fault.statuscode != Fault_None;
return fault.secondstage;

// LSInstructionSyndrome()

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>;
carry_out = extended_x<shift-1>;
return (result, carry_out);

Library pseudocode for shared/functions/common/Abs

// 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

// 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

// 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

// CountLeadingSignBits()
// ======================
integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);

Library pseudocode for shared/functions/common/CountLeadingZeroBits

// 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 vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// ============================
bits(size) Elem<bits(N) vector, integer e]
    return Elem[vector, e, size];

// Elem[] - assignment form
// ========================
Elem<bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;

// Elem[] - assignment form
// ========================
Elem<bits(N) &vector, integer e] = bits(size) value
    Elem[vector, 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-1 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;

Library pseudocode for shared/functions/common/IsOnes

// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);

Library pseudocode for shared/functions/common/IsZero

// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);

Library pseudocode for shared/functions/common/IsZeroBit

// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';

Library pseudocode for shared/functions/common/LSL

// 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;
Library pseudocode for shared/functions/common/LSL_C

// 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);

Library pseudocode for shared/functions/common/LSR

// 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-1: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(M) 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<M-1>, 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

// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";

Library pseudocode for shared/functions/crc/Poly32Mod2

// 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

// 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<100+: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 FFmul0E(in2<c*8+:8>) EOR FFmul0B(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/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 = (/*...*/);
    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<c*8+:8> = FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR in2<c*8+:8> EOR in3<c*8+:8>;
        out1<c*8+:8> = in0<c*8+:8> EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR in3<c*8+:8>;
        out2<c*8+:8> = in0<c*8+:8> EOR in1<c*8+:8> EOR FFmul02(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>);
        out3<c*8+:8> = FFmul03(in0<c*8+:8>) EOR in1<c*8+:8> EOR in2<c*8+:8> EOR FFmul02(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/AESShiftRows

// 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< 16+: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

// 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 = (/* F E D C B A 9 8 7 6 5 4 3 2 1 0 */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610f6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74de8c6ba461c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162ac3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee46b8902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130cdd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bc5389d928f40a351<127:0> :
        /*6*/ 0xa8f3c07f02f94585334d43fbaaeed<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1f20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b2c8309<127:0> :
        /*3*/ 0x75b227ebe2801207aa059618c323c704<127:0> :
        /*2*/ 0x1531d871fe5a534ccf73f362693f9db7<127:0> :
        /*1*/ 0xc872a49cafa2d4adfo759fa7d9c82ca<127:0> :
        /*0*/ 0x76abd77fe2b670138c566bf27b777c63<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 3182


Library pseudocode for shared/functions/crypto/FFmul09

// 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> : /*
    /*E*/ 0xD6DFC4DF2FBE0E99E978C85B83ABA81<127:0> :
    /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
    /*C*/ 0xEDE4FF6C900B2A5ACB7BE8188939A<127:0> :
    /*B*/ 0x303922B141D060F78716A635C554E47<127:0> :
    /*A*/ 0xA0A9B2BB84D969FE8E1FAF3CC5DED7<127:0> :
    /*9*/ 0x0B0219102F263D4434A5158676E757C<127:0> :
    /*8*/ 0x9B92B908BF6ADA43DA0C18F7EE5EC<127:0> :
    /*7*/ 0xAAA3B818E79C95E2EB0F906C6F4D4D<127:0> :
    /*6*/ 0x3332B211E170C5727B606955F444D<127:0> :
    /*5*/ 0x9198838AB5BCA79AED90CB2FDF4EFE6<127:0> :
    /*4*/ 0x0108131252373E4905B26647F7F6<127:0> :
    /*3*/ 0xDCD5CEC7F81F9A9498068FB0092AB<127:0> :
    /*2*/ 0x4C555E768617A7304D16F209232B<127:0> :
    /*1*/ 0xE7EEF55C3CA01D80AF6B0B8829990<127:0> :
    /*0*/ 0x777E656C535A41483F632624182D900<127:0> ;
return FFmul_09<UInt(b)*8+:8>;

Library pseudocode for shared/functions/crypto/FFmul0B

// 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*/ 0xA3A8B5E8F849992F806EDE0D7DEC1CA<127:0> : /*
    /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
    /*D*/ 0x3DD33C54FFEC98888969D6CA7B1A8<127:0> :
    /*C*/ 0xB6637E7444F5259303262D1C170A01<127:0> :
    /*B*/ 0x555E434879726F480061B8012A373<127:0> :
    /*A*/ 0xE5EF3F8C982D04B06A8A0199878C<127:0> :
    /*9*/ 0x2E253B390209141F767066B5A514C47<127:0> :
    /*8*/ 0x9E98B382B2894A4F6C6CDB0EEA1FC7F<127:0> :
    /*7*/ 0x54542A97873665650C71A1202B363D<127:0> :
    /*6*/ 0x64E4FF29C83D658BC7AAA19098886D<127:0> :
    /*5*/ 0x2F2439320308151E777C616A5804D46<127:0> :
    /*4*/ 0x9F9489B23B89A5A6C7C61DDB00BEA1FC7F<127:0> :
    /*3*/ 0xA29B4BF8E58993FAF1EC7E76D0C9C<127:0> :
    /*2*/ 0x12190403E3528234A4155C7666D707B<127:0> :
    /*1*/ 0x0D92DCFC4F5FEE8818A979CADA68B00<127:0> :
    /*0*/ 0x69627F74454E5358313A272C1D16000<127:0> ;
return FFmul_0B<UInt(b)*8+:8>;

Shared Pseudocode Functions Page 3183
Library pseudocode for shared/functions/crypto/FFmul0D

```c
// 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*/ 0x979A8D80A3AEB9B4FFF2E57CB601DC<127:0> : 
    /* E*/ 0x47A45D50737E69642F2235381B16010C<127:0> : 
    /* D*/ 0x2C21363B1815020F44495E537076A67<127:0> : 
    /* C*/ 0xFCF1E6ECB8C5D2DF94998E3A0ADBAB7<127:0> : 
    /* B*/ 0xF07E0EDECE3D49929F88B56A4DBC8B1<127:0> : 
    /* A*/ 0x2A27303D1E13040942F5B5576B6C1G<127:0> : 
    /* 9*/ 0x414C5B567578F6229243331E01070A<127:0> : 
    /* 8*/ 0x919CBB86A5A88BF2F94E3EEDCD07DA<127:0> : 
    /* 7*/ 0x4D0575A7974636E25283F32110C80B6<127:0> : 
    /* 6*/ 0x9D90878A9A43BF5F8F82E1C8CD6<127:0> : 
    /* 5*/ 0xF6FBECE1C2CFD8D59E38499A7B0BD<127:0> : 
    /* 4*/ 0x262B31121F08054E435597A77606D<127:0> : 
    /* 3*/ 0x202D3714190E03484525F7C71666B<127:0> : 
    /* 2*/ 0xF0FD9E7C4C98D9399B52BFACA1B68B<127:0> : 
    /* 1*/ 0x9B96B1B4AFA2B5BFBF3EE9E4C7ADD0D<127:0> : 
    /* 0*/ 0x4B46515C7F72656B23E9391A0D0<127:0> 
    
); return FFmul_0D<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0E

```c
// 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*/ 0x8D83919FB5BBA9A7FDF3E1F5C5CB09D7<127:0> : 
    /* E*/ 0x6D63717F555B49471D13010F25283937<127:0> : 
    /* D*/ 0x56584A446E0727C26283A341E10020C<127:0> : 
    /* C*/ 0x866BAA48E80929CC6C8D04F0E02E<127:0> : 
    /* B*/ 0x202E34C818640A50E4C426866747A<127:0> : 
    /* A*/ 0xC0EDC9F9FE1AF0BECA2886949A<127:0> : 
    /* 9*/ 0xFDFB5E769C3D0DF18859799B3B0AFA1<127:0> : 
    /* 8*/ 0x181507923203F316B657779535D4F41<127:0> : 
    /* 7*/ 0x8CC200E4FAE8E66BC8A0E848A996<127:0> : 
    /* 6*/ 0x2C22303E141A08065C5240E644A7876<127:0> : 
    /* 5*/ 0x17190B052F21333D67697B55F15434D<127:0> : 
    /* 4*/ 0xFF9F9E8E5FC1D3D87899B958B1A3AD<127:0> : 
    /* 3*/ 0x616FD73597545B111F0D329273538B<127:0> : 
    /* 2*/ 0x818F9D9389B7A5ABF1F9FED3C9C7D5D8<127:0> : 
    /* 1*/ 0xB4BA6A8828C9E90ACA4D06B8F2FEE80E<127:0> : 
    /* 0*/ 0x5A544648626CE702A243638121C0E0<127:0> 
); return FFmul_0E<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/HaveAESExt

```c
// 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 = 0xd690e9fece13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491e5194c02d4a8b1347381212b1f449187c8d3f2602a5f575b144b7e55dfba0045be;
    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*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);
Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

// ClearExclusiveMonitors()
// ========================
// Clear the local Exclusives monitor for the executing PE.
ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

// 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();

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

// 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);

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

// 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);

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

// 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);

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

// 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);

Library pseudocode for shared/functions/exclusive/ProcessorID

// Return the ID of the currently executing PE.
integer ProcessorID();

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

// AArch32.HaveHPDExt()
// ====================
boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

// AArch64.HaveHPDExt()
// ====================
boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
Library pseudocode for shared/functions/extension/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";

Library pseudocode for shared/functions/extension/Have52BitVAExt

// 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";

Library pseudocode for shared/functions/extension/HaveAArch32BF16Ext

// 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";

Library pseudocode for shared/functions/extension/HaveAArch32Int8MatMulExt

// 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";

Library pseudocode for shared/functions/extension/HaveAtomicExt

// HaveAtomicExt()
// ===============

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);

Library pseudocode for shared/functions/extension/HaveBF16Ext

// 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";

Library pseudocode for shared/functions/extension/HaveBTIExt

// HaveBTIExt()
// =-----------
// Returns TRUE if support for Branch Target Indentification is implemented.

boolean HaveBTIExt()
    return HasArchVersion(ARMv8p5);
Library pseudocode for shared/functions/extension/HaveBlockBBM

// HaveBlockBBM()
// ==============
// Returns TRUE if support for changing block size without requiring break-before-make is implemented.

boolean HaveBlockBBM()
    return HasArchVersion(ARMv8p4);

Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

// HaveCommonNotPrivateTransExt()
// ==============================

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);

Library pseudocode for shared/functions/extension/HaveDGHExt

// HaveDGHExt()
// =============
// Returns TRUE if Data Gathering Hint instruction support is implemented, and FALSE otherwise.

boolean HaveDGHExt()
    return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";

Library pseudocode for shared/functions/extension/HaveDITExt

// HaveDITExt()
// ============

boolean HaveDITExt()
    return HasArchVersion(ARMv8p4);

Library pseudocode for shared/functions/extension/HaveDOTPExt

// 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");

Library pseudocode for shared/functions/extension/HaveDoPD

// 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";

Library pseudocode for shared/functions/extension/HaveDoubleFaultExt

// HaveDoubleFaultExt()
// ====================

boolean HaveDoubleFaultExt()
    return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB());
Library pseudocode for shared/functions/extension/HaveDoubleLock

```java
// 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";
```

Library pseudocode for shared/functions/extension/HaveE0PDExt

```java
// HaveE0PDExt()
// =============
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.
boolean HaveE0PDExt()
return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveEMPAMExt

```java
// HaveEMPAMExt()
// ===============
// Returns TRUE if Enhanced MPAM is implemented, and FALSE otherwise.
boolean HaveEMPAMExt()
return (HasArchVersion(ARMv8p6) &&
HaveMPAMExt() &&
boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension");
```

Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```java
// HaveExtendedCacheSets()
// ==============
// boolean HaveExtendedCacheSets()
// =============
return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```java
// HaveExtendedECDebugEvents()
// =============
// boolean HaveExtendedECDebugEvents()
// =============
return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```java
// HaveExtendedExecuteNeverExt()
// =============
// boolean HaveExtendedExecuteNeverExt()
// =============
return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveFCADDExt

```java
// HaveFCADDExt()
// =============
// boolean HaveFCADDExt()
// =============
return HasArchVersion(ARMv8p3);
```
Library pseudocode for shared/functions/extension/HaveFGTExt

```
// HaveFGTExt()
// ============
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.

boolean HaveFGTExt()
  return HasArchVersion(ARMv8p6) && !ELUsingAArch32(EL2);
```

Library pseudocode for shared/functions/extension/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()
// ===============

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/HaveFlagFormatExt

```
// HaveFlagFormatExt()
// ===================
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFlagFormatExt()
  return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()
// =======================
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
  return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveFrintExt

```
// HaveFrintExt()
// ==============
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()
  return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
  return HasArchVersion(ARMv8p1);
```
// HaveIDSExt()
// ============
// Returns TRUE if ID register handling feature is implemented.
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");

// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.
boolean HaveMPAMExt()
  return (HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has MPAM extension");

// HaveMTEExt()
// ============
// Returns TRUE if MTE implemented, and FALSE otherwise.
boolean HaveMTEExt()
  if !HasArchVersion(ARMv8p5) then
    return FALSE;
  return boolean IMPLEMENTATION_DEFINED "Has MTE extension";

// HaveNV2Ext()
// ===========
// Returns TRUE if Enhanced Nested Virtualization is implemented.
boolean HaveNV2Ext()
  return (HasArchVersion(ARMv8p4) && HaveNVExt() && boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");
Library pseudocode for shared/functions/extension/HaveNVExt

```java
// HaveNVExt()
// ===========
// Returns TRUE if Nested Virtualization is implemented.
boolean HaveNVExt()
    return HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization";
```

Library pseudocode for shared/functions/extension/HaveNoSecurePMUDisableOverride

```java
// HaveNoSecurePMUDisableOverride()
// ================================
boolean HaveNoSecurePMUDisableOverride()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveNoninvasiveDebugAuth

```java
// HaveNoninvasiveDebugAuth()
// ===========================
// Returns TRUE if the Non-invasive debug controls are implemented.
boolean HaveNoninvasiveDebugAuth()
    return !HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HavePANExt

```java
// HavePANExt()
// =============
boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```java
// HavePageBasedHardwareAttributes()
// =================================
boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HavePrivATExt

```java
// HavePrivATExt()
// ===============
boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```java
// HaveQRDMLAHExt()
// ================
boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```
Library pseudocode for shared/functions/extension/HaveRASExt

```java
// HaveRASExt()
// ============

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) || boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

Library pseudocode for shared/functions/extension/HaveRNG

```java
// 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

```java
// 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

```java
// 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

```java
// HaveSecureEL2Ext()
// ==================

// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

```java
// 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

```java
// HaveSelfHostedTrace()
// =====================

boolean HaveSelfHostedTrace()
    return HasArchVersion(ARMv8p4);
```
// HaveSmallPageTblExt()
// =====================
// Returns TRUE if Small Page Table Support is implemented.

boolean HaveSmallPageTblExt()
return HasArchVersion(ARMv8p4) & boolean IMPLEMENTATION_DEFINED "Has Small Page Table extension";

// HaveStage2MemAttrControl()
// ==========================
// Returns TRUE if support for Stage2 control of memory types and cacheability attributes is implemented.

boolean HaveStage2MemAttrControl()
return HasArchVersion(ARMv8p4);

// HaveStatisticalProfiling()
// ==========================

boolean HaveStatisticalProfiling()
return HasArchVersion(ARMv8p2);

// 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";
Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```c
// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveUA16Ext

```c
// HaveUA16Ext()
// =============
// Returns TRUE if extended unaligned memory access support is implemented, and FALSE otherwise.

boolean HaveUA16Ext()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveUAOExt

```c
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveVirtHostExt

```c
// HaveVirtHostExt()
// =================

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```c
// 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);
```
Library pseudocode for shared/functions/float/bfloat/BFAdd

// BFAdd()
// ========
// Single-precision add following BFloat16 computation behaviors.

bits(32) BFAdd(bits(32) op1, bits(32) 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 && 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);
    return result;

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;
return result;
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) = BFPack(op1);
    (type2,sign2,value2) = BFPack(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);
    return result;
```
// 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;
// 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;
    else    // no SNaN for BF16 arithmetic
        fptype = FPType_QNaN; value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);
else
    if sign == '1' then value = -value;

return (fptype, sign, value);
// FPConvertBF()  
// =============  
// Converts a single-precision OP to BFloat16 value with rounding controlled by Rounding.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr, FPRounding rounding)
  bits(32) result;    // BF16 value in top 16 bits

  // Unpack floating-point operand optionally with flush-to-zero.
  (fptype,sign,value) = FPUnpack(op, fpcr);

  if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
    if fpcr.DN == '1' then
      result = FPDefaultNaN();
    else
      result = FPConvertNaN(op);
  elseif fptype == FPTYPE_Infinity then
    result = FPInfinity(sign);
  elseif fptype == FPTYPE_Zero then
    result = FPZero(sign);
  else
    result = FPRoundCVBF(value, fpcr, rounding);

  // 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));

// FPRoundCVBF()
// =============
// Converts a real number OP into a BFloat16 value using the supplied rounding mode RMODE.

bits(32) FPRoundCVBF(real op, FPCRType fpcr, FPRounding rounding)
  boolean isbfloat = TRUE;
  return FPRoundBase(op, fpcr, rounding, isbfloat);
Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```plaintext
// 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

```plaintext
// FPAbs()
// ========

bits(N) FPAbs(bits(N) op)
assert N IN {16,32,64};
return '0' : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd

```plaintext
// 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);
        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==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
  result = '0011';
  if type1==FPType_SNaN || type2==FPType_SNaN || signal_nans then
    FPProcessException(FPExc_InvalidOp, fpcr);
  else
    // All non-NaN cases can be evaluated on the values produced by FPUnpack()
    if value1 == value2 then
      result = '0110';
    elseif value1 < value2 then
      result = '1000';
    else  // value1 > value2
      result = '0010';
  return result;

Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

// 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==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==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);
  return result;

Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

// 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==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==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);
  return result;
Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

// 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==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==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);
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.
(fptype,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);
    endif
else
    result = FPConvertCV(value, fpcr, rounding);
endif
// 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<51-1>;

// 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' return FPRounding_TIEAWAY; // A
    when '01' return FPRounding_TIEEVEN; // N
    when '10' return FPRounding_POSINF; // P
    when '11' return FPRounding_NEGINF; // M

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

// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()
assert N IN {16,32,64};
custom integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
custom integer F = N - (E + 1);
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

// 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();
    FPProcessException(FPExc_InvalidOp, fpcr);
  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);
  return result;

Library pseudocode for shared/functions/float/fpexc/FPExc

enumeration FPExc
  {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
   FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

// FPInfinity()
// =============

bits(N) FPInfinity(bit sign)
assert N IN {16,32,64};
custom integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
custom 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

// 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)
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
  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.
    result = FPRound(value, 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

// 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);

// treat a single quiet-NaN as -Infinity
if type1 == FPTYPE_QNaN && type2 != FPTYPE_QNaN then
    op1 = FPIInfinity('1');
elsif type1 != FPTYPE_QNaN && type2 == FPTYPE_QNaN then
    op2 = FPIInfinity('1');
return FPMax(op1, op2, fpcr);

Library pseudocode for shared/functions/float/fpmin/FPMin

// FPMin()
// ========

bits(N) FPMin(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
    if value1 < value2 then
        (fptype,sign,value) = (type1,sign1,value1);
    else
        (fptype,sign,value) = (type2,sign2,value2);
    if fptype == FPTYPE_INFINITY then
        result = FPIInfinity(sign);
    elsif fptype == FPTYPE_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.
        result = FPRound(value, fpcr);
    return result;

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

// 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);

// Treat a single quiet-NaN as +Infinity
if type1 == FPTYPE_QNaN && type2 != FPTYPE_QNaN then
    op1 = FPIInfinity('0');
elsif type1 != FPTYPE_QNaN && type2 == FPTYPE_QNaN then
    op2 = FPIInfinity('0');
return FPMin(op1, op2, fpcr);
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);
return result;
// FPMulAdd()
// =========
// // Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
assert N IN {16,32,64};
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) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

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.
    if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
        result = FPDefaultNaN();
        FPProcessException(FPEXC_INVALIDOP, fpcr);

    // Other cases involving infinities produce an infinity of the same sign.
    elsif (infA && signA == '0') || (infP && signP == '0') then
        result = FPIInfinity('0');
    elsif (infA && signA == '1') || (infP && signP == '1') then
        result = FPIInfinity('1');

    // Cases where the result is exactly zero and its sign is not determined by the
    // rounding mode are additions of same-signed zeros.
    elsif 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);

    return result;
Library pseudocode for shared/functions/float/fpmladdh/FPMulAddH

// FPMulAddH()
// ===========

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
  assert N IN {32,64};
  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 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);
    signP = sign1 EOR sign2;
    infP = inf1 || inf2;
    zeroP = zero1 || zero2;
    // Determine sign and type product will have if it does not cause an Invalid
    // Operation.
    if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
      result = FPDefaultNaN();
      FPProcessException(FPExc_InvalidOp, fpcr);
    // Other cases involving infinities produce an infinity of the same sign.
    elsif (infA && signA == '0') || (infP && signP == '0') then
      result = FPIInfinity('0');
    elsif (infA && signA == '1') || (infP && signP == '1') then
      result = FPIInfinity('1');
    // Cases where the result is exactly zero and its sign is not determined by the
    // rounding mode are additions of same-signed zeros.
    elsif 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);
      return result;
Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

// FPProcessNaNs3H()
// =================

(bool, 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;
if type1 == FPType_SNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_SNaN then
done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
elsif type3 == FPType_SNaN then
done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
elsif type1 == FPType_QNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPType_QNaN then
done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
elsif type3 == FPType_QNaN then
done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
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 = FPZero(sign1 EOR sign2);
else
result = FPRound(value1*value2, fpcr);
return result;

Library pseudocode for shared/functions/float/fpneg/FPNeg

// FPNeg()
// ========

bits(N) FPNeg(bits(N) op)
assert N IN {16,32,64};
return NOT(op<N-1>) : op<N-2:0>;
Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```plaintext
// 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);
    return sign : exp : frac;
```

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
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```plaintext
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)
    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';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
```
**Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs**

```plaintext
// 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. 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)
```

```plaintext
assert N IN {16,32,64};
if type1 == FPTYPE_SNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPTYPE_SNaN then
done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elsif type1 == FPTYPE_QNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPTYPE_QNaN then
done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
else
done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);
```

**Library pseudocode for shared/functions/float/fpprocessnans3/FPProcessNaNs3**

```plaintext
// 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. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPTYPE type1, FPTYPE type2, FPTYPE type3,
                                bits(N) op1, bits(N) op2, bits(N) op3,
                                FPCRType fpcr)
```

```plaintext
assert N IN {16,32,64};
if type1 == FPTYPE_SNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPTYPE_SNaN then
done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elsif type3 == FPTYPE_SNaN then
done = TRUE; result = FPProcessNaN(type3, op3, fpcr);
elsif type1 == FPTYPE_QNaN then
done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
elsif type2 == FPTYPE_QNaN then
done = TRUE; result = FPProcessNaN(type2, op2, fpcr);
elsif type3 == FPTYPE_QNaN then
done = TRUE; result = FPProcessNaN(type3, op3, fpcr);
else
done = FALSE; result = Zeros(); // 'Don't care' result
return (done, result);
```
FPRecipEstimate()  
FPRecipEstimate(bits(N) operand, FPCRType fpcr)  
assert N IN {16,32,64};  
(fptype,sign,value) = FPUnpack(operand, fpcr);  
if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then  
result = FPProcessNaN(fptype, operand, fpcr);  
elseif fptype == FPTYPE_Infinity then  
result = FPZero(sign);  
elseif fptype == FPTYPE_Zero then  
result = FPInfinity(sign);  
FPProcessException(FPExc_DivideByZero, fpcr);  
elseif (  
(N == 16 && Abs(value) < 2.0^-16) ||  
(N == 32 && Abs(value) < 2.0^-128) ||  
(N == 64 && Abs(value) < 2.0^-1024)  
) then  
case FPRoundingMode(fpcr) 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);  
FPProcessException(FPExc_Overflow, fpcr);  
FPProcessException(FPExc_Inexact, fpcr);  
elseif ((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);  
if UsingAArch32() then  
FPSCR.UFC = '1';  
else  
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>;  
  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 = UInt('1':fraction<51:44>);  
  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 representing a fixed-point number in range [0.5..1.0)
estimate = RecipEstimate(scaled);

// estimate is in the range 256..511 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.

fraction = estimate<7:0> : Zeros(44);
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;

Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

// Compute estimate of reciprocal of 9-bit fixed-point number
// // a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// // result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipEstimate(integer a)
assert 256 <= a && a < 512;
a = a*2+1; // round to nearest
integer b = (2 ^ 19) DIV a;
r = (b+1) DIV 2; // round to nearest
assert 256 <= r && r < 512;
return r;
```
// 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();

   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;

   (fptype,sign,value) = FPUnpack(op, fpcr);
   if fptype == FPTyp_SNaN || fptype == FPTyp_QNaN then
      result = FPProcessNaN(fptype, op, fpcr);
   else
      if IsZero(exp) then // Zero and denormals
         result = sign:max_exp:frac;
      else // Infinities and normals
         result = sign:NOT(exp):frac;
   return result;
```
Library pseudocode for shared/functions/float/fround/FPRound
// 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 isbfloat = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat);

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat)
    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 && isbfloat 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;

    // Deal with flush-to-zero.
    if ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) && exponent < minimum_exp then
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            FPSR.UFC = '1';
        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);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
        FPProcessException(FPExc_Underflow, fpcr);
    // Round result according to rounding mode.
    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 aka Von Neumann rounding
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);
    FPProcessException(FPExc_Overflow, fpcr);
    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);
    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
  FPProcessException(FPExc_Inexact, fpcr);

return result;

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
return FPRound(op, fpcr, FPRoundingMode(fpcr));

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 isbfloat = FALSE;
return FPRoundBase(op, fpcr, rounding, isbfloat);

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);

Library pseudocode for shared/functions/float/fproundint/FPRoundInt

// FPRoundInt()
// ============

// Round OP to nearest integral floating point value using rounding mode ROUNDING. If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
assert rounding != FPRounding_ODD;
assert N IN {16,32,64};

// Unpack using FPCR to determine if subnormals are flushed-to-zero
(fptype,sign,value) = FPUnpack(op, fpcr);

if fptype == FPType_SNaN || fptype == FPType_QNaN then
result = FPProcessNaN(fptype, op, fpcr);
elsif fptype == FPType_Infinity then
result = FPInfinity(sign);
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_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;
// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCRTyp 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);
// Unpack using FPCR to determine if subnormals are flushed-to-zero
(fptype, sign, value) = FPUnpack(op, fpcr);
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_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;
    if int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1) 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);
        // Generate inexact exceptions
        if error != 0.0 then
            FPProcessException(FPExc_Inexact, fpcr);
    return result;
Library pseudocode for shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

// FPRSqrtEstimate()
// ===============

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
assert N IN {16,32,64};
(fptype,sign,value) = FPUnpack(operand, fpcr);
if fptype == FPTYPE_SNAN || fptype == FPTYPE_QNAN then
    result = FPProcessNaN(fptype, operand, fpcr);
elsif fptype == FPTYPE_ZERO then
    result = FPInfinity(sign);
    FPProcessException(FPExc_DivideByZero, fpcr);
elsif sign == '1' then
    result = FPDefaultNaN();
    FPProcessException(FPExc_InvalidOp, fpcr);
elsif fptype == FPTYPE_INFINITY 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;
        end
        fraction = fraction<50:0> : '0';
    if exp<0> == '0' then
        scaled = UInt('1':fraction<51:44>);
    else
        scaled = UInt('01':fraction<51:45>);
    end
    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);
    // estimate is in the range 256..511 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 result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
        when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);
    return result;
Library pseudocode for shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

// Compute estimate of reciprocal square root of 9-bit fixed-point number
//
// a is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a)
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
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) do
r = (b+1) DIV 2; // round to nearest
assert 256 <= r && r < 512;
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();
    FProcessException(FPExc_InvalidOp, fpcr);
else
    result = FPRound(Sqrt(value), fpcr);
return result;
// 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);
    end
    return result;
end

// 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);
return sign : exp : frac;

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);
return sign : exp : frac;
/ **FPToFixed()**

// Convert N-bit precision floating point OP to M-bit fixed point with FBITS fractional bits, controlled by UNSIGNED and ROUNDDING.

```
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;

// Unpack using fpcr to determine if subnormals are flushed-to-zero fptype,sign,value = FPUnpack(op, fpcr);

// 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 (result, overflow) = SatQ(int_result, M, unsigned);
if overflow then FPProcessException(FPExc_InvalidOp, fpcr);
elsif 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, FPCRTypetyp, boolean Is64)

    assert M == 64 && N == 32;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (fptype, sign, value) = FPUnpack(op, fpcr);

    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);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
    elsif sign == '1' && value == 0.0 then
        Z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        Z = '0';
    if fptype == FPTYPE_Infinity 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);
    return sign : exp : frac;

Library pseudocode for shared/functions/float/fptype/FPType

enumeration FPType {FPTYPE_Nonzero, FPTYPE_Zero, FPTYPE_Infinity,
    FPTYPE_QNaN, FPTYPE_SNaN};
Library pseudocode for shared/functions/float/fpunpack/FPUnpack

// 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) 
  fpcr.AHP = '0';
  (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
  return (fp_type, sign, value);
Library pseudocode for shared/functions/float/fpunpack/FPUnpackBase
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. Status information is
updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)

assert N IN {16,32,64};

if N == 16 then
  sign   = fpval<15>;
  exp16  = fpval<14:10>;
  frac16 = fpval<9:0>;
  if IsZero(exp16) then
    // Produce zero if value is zero or flush-to-zero is selected
    if IsZero(frac16) || fpcr.FZ16 == '1' then
      fptype = FPType_Zero;  value = 0.0;
    else
      fptype = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
  elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
    if IsZero(frac16) then
      fptype = FPType_Infinity;  value = 2.0^1000000;
    else
      fptype = if frac16<9> == '1' then
        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));
  else

elsif N == 32 then
  sign   = fpval<31>;
  exp32  = fpval<30:23>;
  frac32 = fpval<22:0>;
  if IsZero(exp32) then
    // Produce zero if value is zero or flush-to-zero is selected.
    if IsZero(frac32) || fpcr.FZ == '1' then
      fptype = FPType_Zero;  value = 0.0;
    if !IsZero(frac32) then  // Denormalized input flushed to zero
      FPProcessException(FPExc_InputDenorm, fpcr);
    else
      fptype = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
  elsif IsOnes(exp32) then
    if IsZero(frac32) then
      fptype = FPType_Infinity;  value = 2.0^1000000;
    else
      fptype = if frac32<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

elsif N == 64 then
  sign   = fpval<63>;
  exp64  = fpval<62:52>;
  frac64 = fpval<51:0>;
  if IsZero(exp64) then
    // Produce zero if value is zero or flush-to-zero is selected.
    if IsZero(frac64) || fpcr.FZ == '1' then
      fptype = FPType_Zero;  value = 0.0;
    if !IsZero(frac64) then  // Denormalized input flushed to zero
      FPProcessException(FPExc_InputDenorm, fpcr);
    else

Shared Pseudocode Functions
fptype = FPType_Nonzero; value = \(2.0^{-1022} \times (\text{Real} (\text{UInt} (\text{frac64})) \times 2.0^{-52})\);

elsif 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;
    else
        fptype = FPType_Nonzero;
        value = 2.0^{(\text{UInt}(\text{exp64})-1023)} \times (1.0 + \text{Real}(\text{UInt}(\text{frac64})) \times 2.0^{-52});
    
    if sign == '1' then value = -value;
    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';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    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);
    return sign : exp : frac;

Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

// VFPEntendImm()
// ==============

bits(N) VFPEntendImm(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);
    return sign : exp : frac;
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

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 actions
  AccType_ORDEREDATOMICRW,            // Load-AOAcquire and Store-LORelease
  AccType_LIMTEDORDERED,              // Load-LOAcquire and Store-LORelease
  AccType_UNPRIV,                     // Load and store unprivileged
  AccType_IFETCH,                     // Instruction fetch
  AccType_PTW,                        // Page 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
  // Other operations              // to a memory access that uses the EL2 translation
  AccType_DC,                         // Data cache maintenance
  AccType_DC_UNPRIV,                  // Data cache maintenance instruction used at EL1
  AccType_IC,                         // Instruction cache maintenance
  AccType_DCZVA,                      // DC ZVA instructions
  AccType_AT};                        // Address translation

Library pseudocode for shared/functions/memory/AccessDescriptor

type AccessDescriptor is (AccType acctype, boolean transactional, MPAMinfo mpam, boolean page_table_walk, boolean secondstage, boolean s2fs1walk, integer level)
// 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.

type AddressDescriptor is (FaultRecord fault,   // fault.statuscode indicates whether the address is valid
                        MemoryAttributes memattrs,
                        FullAddress paddress,
                        bits(64) vaddress)

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

// BigEndian()
// ===========

// Library pseudocode for shared/functions/memory/AddressDescriptor

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

// boolean BigEndian()
// =============
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 IN {AccType_IFETCH, AccType_IC});
accdesc.page_table_walk = FALSE;
return accdesc;

// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype) transactional = FALSE;
return CreateAccessDescriptor(acctype, transactional);

Library pseudocode for shared/functions/memory/CreateAccessDescriptorPTW

// CreateAccessDescriptorPTW()
// ===========================

AccessDescriptor CreateAccessDescriptorPTW(AccType acctype, boolean secondstage,
boolean s2fs1walk, integer level)
AccessDescriptor accdesc; accdesc.acctype = acctype;
accdesc.transactional = FALSE;
accdesc.mpam = GenMPAMcurEL(acctype IN {AccType_IFETCH, AccType_IC});
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.TBID1 else TCR_EL1.TBID0;  
  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.TBID1 else TCR_EL2.TBID0;  
    else  
      tbi = TCR_EL2.TBI;  
      if HavePACExt() then tbid = TCR_EL2.TBID;  
  when EL3  
    tbi = TCR_EL3.TBI;  
    if HavePACExt() then tbid = TCR_EL3.TBID;  

  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

```c
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_Exclusive,
    Fault_ICacheMaint};
```

Library pseudocode for shared/functions/memory/FaultRecord

```c
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 page 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
    boolean secondstage, // Is a Stage 2 abort
    bits(4) domain, // Domain number, AArch32 only
    bits(2) errortype, // [Armv8.2 RAS] AArch32 AET or AArch64 SET
    bits(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 pmsg)
)
```

Library pseudocode for shared/functions/memory/FullAddress

```c
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 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/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, 
    bit nG, // '0' = Global, '1' = not Global 
    bit(4) domain, // AArch32 only 
    bit GP, // Guarded Page 
    boolean contiguous, // Contiguous bit from page 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)

Library pseudocode for shared/functions/memory/Tag

constant integer LOG2_TAG_GRANULE = 4;
constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;

Library pseudocode for shared/functions/memory/_Mem

// 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;

Library pseudocode for shared/functions/mpam/DefaultMPAMinfo

// DefaultMPAMinfo 
// =============== 
// 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;

Library pseudocode for shared/functions/mpam/DefaultPARTID

constant PARTIDtype DefaultPARTID = 0<15:0>;
constant PMGtype DefaultPMG = 0<7:0>;

// GenMPAMcurEL
// ============
// Returns MPAMinfo for the current EL and security state.
// InD is TRUE instruction access and FALSE otherwise.
// 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(boolean InD)

  bits(2) mpamel;
  boolean validEL;
  boolean securempam;
  if HaveEMPAMExt() then
    boolean secure = IsSecure();
    securempam = MPAM3_EL3.FORCE_NS == '0' && secure;
    if MPAMisDisabled() && (!secure || MPAM3_EL3.SDEFLT == '0') then
      if UsingAArch32() then
        (validEL, mpamel) = ELFromM32(PSTATE.M);
      else
        validEL = TRUE;
        mpamel = PSTATE.EL;
      if validEL then
        return genMPAM(UInt(mpamel), InD, securempam);
    return DefaultMPAMinfo(securempam);
  else
    securempam = IsSecure();
    if HaveMPAMExt() & MPAMisDisabled() then
      if UsingAArch32() then
        (validEL, mpamel) = ELFromM32(PSTATE.M);
      else
        validEL = TRUE;
        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?
    elsif 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';
Library pseudocode for shared/functions/mpam/MPAMisVirtual

// MPAMisVirtual
// =============
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(integer el)
return ( MPAMIDR_EL1.HAS_HCR == '1' &&
( HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0' ) &&
( el == 0 && MPAMHCR_EL2.EL0_VPMEN == '1' ) ||
( el == 1 && MPAMHCR_EL2.EL1_VPMEN == '1' ));

Library pseudocode for shared/functions/mpam/genMPAM

// 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 &&
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;

Library pseudocode for shared/functions/mpam/genMPAMel

// 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 HaveEMPAMExt() 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);
Library pseudocode for shared/functions/mpam/genPARTID

// 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);

Library pseudocode for shared/functions/mpam/genPMG

// 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;
    return DefaultPMG;

Library pseudocode for shared/functions/mpam/getMPAM_PARTID

// getMPAM_PARTID
// ==============
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAM 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;
            end;
        end;
    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;
        end;
    end;
    return partid;
// 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;
}

// 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 REM 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();
        _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();
        _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(32) SPSR[]
    result;
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ   result = SPSR_fiq;
            when M32_IRQ   result = SPSR_irq;
            when M32_Svc   result = SPSR_svc;
            when M32_Monitor   result = SPSR_mon;
            when M32_Abort   result = SPSR_abt;
            when M32_Hyp   result = SPSR_hyp;
            when M32.Undef   result = SPSR_und;
            otherwise     Unreachable();
        end;
    else
        case PSTATE.EL of
            when EL1    result = SPSR_EL1;
            when EL2    result = SPSR_EL2;
            when EL3    result = SPSR_EL3;
            otherwise     Unreachable();
        end;
    end;
    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(32) value
    if UsingAArch32() then
        case PSTATE.M of
            when M32_FIQ   SPSR_fiq = value;
            when M32_IRQ   SPSR_irq = value;
            when M32_Svc   SPSR_svc = value;
            when M32_Monitor   SPSR_mon = value;
            when M32_Abort   SPSR_abt = value;
            when M32_Hyp   SPSR_hyp = value;
            when M32.Undef   SPSR_und = value;
            otherwise     Unreachable();
        end;
    else
        case PSTATE.EL of
            when EL1    SPSR_EL1 = value;
            when EL2    SPSR_EL2 = value;
            when EL3    SPSR_EL3 = value;
            otherwise     Unreachable();
        end;
    end;
    return;
Library pseudocode for shared/functions/system/ArchVersion

```c
enumeration ArchVersion {
    ARMv8p0,
    ARMv8p1,
    ARMv8p2,
    ARMv8p3,
    ARMv8p4,
    ARMv8p5,
    ARMv8p6
};
```

Library pseudocode for shared/functions/system/BranchTargetCheck

```c
// 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

```c
// ClearEventRegister()
// -------------------
// Clear the Event Register of this PE

ClearEventRegister() {
    EventRegister = '0';
    return;
}
```

Library pseudocode for shared/functions/system/ClearPendingPhysicalSError

```c
// Clear a pending physical SError interrupt
ClearPendingPhysicalSError();
```

Library pseudocode for shared/functions/system/ClearPendingVirtualSError

```c
// Clear a pending virtual SError interrupt
ClearPendingVirtualSError();
```
Library pseudocode for shared/functions/system/ConditionHolds

// 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;

Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

ConsumptionOfSpeculativeDataBarrier();

Library pseudocode for shared/functions/system/CurrentInstrSet

// 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;

Library pseudocode for shared/functions/system/CurrentPL

// CurrentPL()
// ===========
PrivilegeLevel CurrentPL()

    return PLoFEL(PSTATE.EL);

Library pseudocode for shared/functions/system/EL0

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

// 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

// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
// 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;
    when M32_FIFO, 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(32) spsr)
if spsr<4> == '0' then                      // AArch64 state
  el = spsr<3:2>;
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)
return (!IsSecureEL2Enabled() || !IsSecureBelowEL3() && HaveVirtHostExt() && !ELUsingAArch32(EL2) &&
  HCR_EL2.E2H == '1' && (el == EL2 || (el == EL0 && HCR_EL2.TGE == '1')));

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

// ELStateUsingAArch32K()
// ======================

(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
elsif el == HighestEL() then
  return (TRUE, FALSE); // This is highest Exception level, so is using AArch64
// Remainder of function deals with the interprocessing cases when highest Exception level is using AArch64
boolean aarch32 = boolean UNKNOWN;
boolean known = TRUE;
aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EEL2 == '0');
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
else
  aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});
if !known then aarch32 = boolean UNKNOWN;
return (known, aarch32);

Library pseudocode for shared/functions/system/ELUsingAArch32

// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
return ELStateUsingAArch32(el, IsSecureBelowEL3());

Library pseudocode for shared/functions/system/ELUsingAArch32K

// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
return ELStateUsingAArch32K(el, IsSecureBelowEL3());

Library pseudocode for shared/functions/system/EndOfInstruction

// Terminate processing of the current instruction.
EndOfInstruction();

Library pseudocode for shared/functions/system/EnterLowPowerState

// PE enters a low-power state
EnterLowPowerState();
Library pseudocode for shared/functions/system/EventRegister

bits(1) EventRegister;

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

// GetPSRFromPSTATE()
// ================
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
    bits(32) 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 spsr<21> = PSTATE.DIT;
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9> = PSTATE.E;
        spsr<8:6> = PSTATE.<A,I,F>; // No PSTATE.D in AArch32 state
        spsr<5> = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW; // bit [4] is the discriminator
        spsr<4:0> = PSTATE.M;
    else // AArch64 state
        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()
// ===============
// Return 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()
// ================
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;

Library pseudocode for shared/functions/system/HaveAnyAArch64

// HaveAnyAArch64()
// ================
// Return TRUE if AArch64 state is supported at any Exception level

boolean HaveAnyAArch64()
    return !HighestELUsingAArch32();

Library pseudocode for shared/functions/system/HaveEL

// HaveEL()
// =========
// Return TRUE if Exception level 'el' is supported

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_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();
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(32) 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

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

InstructionSynchronizationBarrier();

Library pseudocode for shared/functions/system/InterruptPending

// InterruptPending()
// ================
// Return TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise

boolean InterruptPending()
{
    return IsPhysicalSErrorPending() || IsVirtualSErrorPending();
}

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

// Return TRUE if a physical SError interrupt is pending

boolean IsPhysicalSErrorPending();

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;
    else
        return IsSecureBelowEL3();
    endif

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.
// This 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";
    endif
// IsSecureEL2Enabled()
// ===============
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && HaveSecureEL2Ext() then
        if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
            return TRUE;
        else
            return FALSE;
    else
        return IsSecure();
    else
        return FALSE;

// IsVirtualSErrorPending
// ================
// Return TRUE if a virtual SError interrupt is pending

boolean IsVirtualSErrorPending();

// Mode_Bits
// =========

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};
**Library pseudocode for shared/functions/system/ProcState**

```plaintext
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 (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 [AArch32 only]  
  bits (8) IT,      // If-then bits, RES0 in CPSR [AArch32 only]  
  bits (1) J,       // J bit, RES0 [AArch32 only]  
  bits (1) T,       // T32 bit, RES0 in CPSR [AArch32 only]  
  bits (1) E,       // Endianness bit [AArch32 only]  
  bits (5) M        // Mode field [AArch32 only] )
```

**Library pseudocode for shared/functions/system/RestoredITBits**

```plaintext
// RestoredITBits()  
// ================  
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(32) 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**

```plaintext
type SCRType;
```
SCRType SCR_GEN[]

// SCR.Gen[]
// =========

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if HighestELUsingAArch32() then
        r = ZeroExtend(SCR);
    else
        r = ZeroExtend(SCR_EL3);
    return r;

SendEvent();

Library pseudocode for shared/functions/system/SendEventLocal

// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed
SendEventLocal()
    EventRegister = '1';
    return;
// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = 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 HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
    // 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 SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(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.Q = spsr<27>;
        PSTATE.IT = RestoredITBits(spsr);
        ShouldAdvanceIT = FALSE;
        if HaveDITExt() then PSTATE.DIT = (if Restarting() then spsr<24> else spsr<21>);
        PSTATE.GE = spsr<19:16>;
        PSTATE.E = spsr<9>;
        PSTATE.<A,I,F> = spsr<8:6>;
        if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
        PSTATE.SP = spsr<0>;
        if HavePANExt() then PSTATE.PAN = spsr<22>;
        if HaveMTEExt() then PSTATE.TCO = spsr<25>;
        if HaveDITExt() then PSTATE.DIT = spsr<24>;
        if HaveUAOExt() then PSTATE.UAO = spsr<23>;
        if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
        PSTATE.<D,A,I,F> = spsr<9:6>;
    else // AArch64 state
        PSTATE.<Q,IT,GE,E,T> in AArch64 state
    return;

Library pseudocode for shared/functions/system/ShouldAdvanceIT

boolean ShouldAdvanceIT;

Library pseudocode for shared/functions/system/SpeculationBarrier

SpeculationBarrier();

Library pseudocode for shared/functions/system/SynchronizeContext

SynchronizeContext();

Library pseudocode for shared/functions/system/SynchronizeErrors

// Implements the error synchronization event.
SynchronizeErrors();
// Take any pending unmasked physical SError interrupt
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);

// Take any pending unmasked physical SError interrupt or unmasked virtual SError interrupt.
TakeUnmaskedSErrorInterrupts();

bits(32) ThisInstr();

integer ThisInstrLength();

assert FALSE;

// UsingAArch32
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.
boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;

// WaitForEvent
// ==============
// PE suspends its operation and enters a low-power state
// if the Event Register is clear when the WFE is executed
WaitForEvent()  
    if EventRegister == '0' then
        EnterLowPowerState();
    return;

// WaitForInterrupt
// ==============
// PE suspends its operation to enter a low-power state
// until a WFI wake-up event occurs or the PE is reset
WaitForInterrupt()
    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_WBOVERLAPLD return Constraint_WBSUPPRESS; // return loaded value
        when Unpredictable_WBOVERLAPST return Constraint_NONE; // store pre-writeback value
        when Unpredictable_LPPOVERLAP 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_INSTRDEVICE return Constraint_FALSE; // Do not take a fault
        when Unpredictable_RESCPACR return Constraint_UNKNOWN; // Map to UNKNOWN value
        when Unpredictable_RESCAIR return Constraint_UNKNOWN; // Map to UNKNOWN value
        when Unpredictable_RESEXCR return Constraint_UNKNOWN; // Map to UNKNOWN value
        when Unpredictable_RESDACR return Constraint_UNKNOWN; // Map to UNKNOWN value
        when Unpredictable_RESRPRR 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_LARGEIPA return Constraint_FORCE; // Map to the limit 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_WPMASKANDBAS return Constraint_FAULT; // Watchpoint disabled
        when Unpredictable_WPASCONTIGUOUS return Constraint_FALSE; // Watchpoint disabled
        when Unpredictable_RESWPMASK return Constraint_FALSE; // Watchpoint disabled
        when Unpredictable_WPBASEDISABED return Constraint_FALSE; // Watchpoint disabled
        when Unpredictable_WPMASKEDBITS return Constraint_FALSE; // Watchpoint disabled
        when Unpredictable_RESBPWPCTRL return Constraint_DISABLED; // Breakpoint/watchpoint disabled
        when Unpredictable_BPNOTIMPL return Constraint_DISABLED; // Breakpoint disabled
        Shared Pseudocode Functions
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; // No match
when Unpredictable_RESTARTALIGNPC return Constraint.False; // Do not force alignment
when Unpredictable_RESTARTZEROUPPERPC return Constraint.True; // Force zero extension
when Unpredictable_ZEROPPER return Constraint.True; // zero top halves of X registers
when Unpredictable_ERETZEROUPPERPC 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_AFUPDATE return Constraint.True; // AF update for alignment or permission fault
when Unpredictable_IESBinDebug return Constraint.True; // Use SCTLR[].IESB in Debug state
when Unpredictable_BADPMSFCR return Constraint.True; // Bad settings for PMSFCR_EL1/PMSEVR_EL1/PMSLATFR_EL1
when Unpredictable_ZEROBTYPE return Constraint.True; // Save BTYPE in SPSR_ELX/DPSR_EL0 as '00'
when Unpredictable_CLEARERRITEZERO return Constraint.False; // Clearing sticky errors when instruction in flight

**Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits**

```plaintext
// 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)
```

```plaintext
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
```

Shared Pseudocode Functions
Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

// 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);
}

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

// 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
}

Library pseudocode for shared/functions/unpredictable/Constraint

enumeration Constraint    {// General
    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,
    Constraint_COND,              // Instruction executes with additional decode
    Constraint_ADDITIONAL_DECODE,
    Constraint_WBSUPPRESS, Constraint_FAULT, // Instruction executes with additional decode
    Constraint_FORCE, Constraint_FORCENOSLCHECK};
Library pseudocode for shared/functions/unpredictable/Unpredictable
Unpredictable WBOVERLAPLD,
Unpredictable WBOVERLAPST,
Unpredictable LDPOVERLAP,
Unpredictable BASEOVERLAP,
Unpredictable DATAOVERLAP,
Unpredictable_DEVPAGE2,
Unpredictable_INSTRDEVICE,
Unpredictable_RESCPACR,
Unpredictable_RESDACR,
Unpredictable_RESDTnSZ,
Unpredictable_LARGEIPA,
Unpredictable_ESRCONDPASS,
Unpredictable_ILZEROIT,
Unpredictable_ILZEROT,
Unpredictable_BPVECTORCATCHPRI,
Unpredictable_VCMATCHHALF,
Unpredictable_VCMATCHDAPA,
Unpredictable_WPMASKANDBAS,
Unpredictable_WPBASCONTIGUOUS,
Unpredictable_RESWPMASK,
Unpredictable_WPMASKEDBITS,
Unpredictable_RESERVEDCTRL,
Unpredictable_BPNOTIMPL,
Unpredictable_RESERVEDTYPE,
Unpredictable_RESERVEDTYPE,
Unpredictable_BPMATCHHALF,
Unpredictable_BPMISMATCHHALF,
Unpredictable_RESTARTALIGNPC,
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,
// 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,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITEZERO);
// AdvSIMDExpandImm()
// ================

bits(64) AdvSIMDExpandImm(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(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            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);
            else
                imm64 = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
            endif
        if cmode<0> == '0' && op == '1' then
            imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
            imm64 = Replicate(imm32, 2);
        if cmode<0> == '1' && op == '0' then
            imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);
    return imm64;

Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

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<N-1:0>, saturated);

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
assert N IN {16,32};
if operand<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)
    case N of
        when 16 estimate = RecipSqrtEstimate(UInt(operand<15:7>));
        when 32 estimate = RecipSqrtEstimate(UInt(operand<31:23>));
    // result is in the range 0x80000000 .. 0x7fff0000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);
return result;
// Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
assert N IN {16,32};
if operand<15:0> == '0' then // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
    result = Ones(N);
else // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)
    // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
    case N of
        when 16 estimate = RecipEstimate(UInt(operand<15:7>));
        when 32 estimate = RecipEstimate(UInt(operand<31:23>));
    // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
    result = estimate<8:0> : Zeros(N-9);
return result;

// Library pseudocode for shared/functions/vector/UnsignedSatQ

// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
if i > 2^N - 1 then
    result = 2^N - 1;  saturated = TRUE;
elsif i < 0 then
    result = 0;  saturated = TRUE;
else
    result = i;  saturated = FALSE;
return (result<8:0>, saturated);

// Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

// 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());
    return (!TraceAllowed() || !HaveEL(EL2));

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 (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP);
            case TS_el2 of
            when '00' /* falls through to check TRFCR_EL1.TS */
            when '01' return TimeStamp_Virtual;
            when '11' return TimeStamp_Physical;
            otherwise Unreachable();
                /* ConstrainUnpredictableBits removes this case */
            TS_el1 = TRFCR_EL1.TS;
            if TS_el1 == '0' then (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP);
            case TS_el1 of
            when '01' return TimeStamp_Virtual;
            when '11' return TimeStamp_Physical;
            otherwise Unreachable();
                /* ConstrainUnpredictableBits removes this case */
        else
            return TimeStamp_CoreSight;

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)

    MemAttrHints result;
    apply_force_writeback = HaveStage2MemAttrControl() && HCR_EL2.FWB == '1';

    if apply_force_writeback then
        if 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 s2descattrs == '01' || s1desc.attrs == '01' then
            result.attrs = bits(2) UNKNOWN;  // Reserved
        elsif s2descattrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
            result.attrs = MemAttr_NC;  // Non-cacheable
        elsif s2descattrs == 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 HaveStage2MemAttrControl() && HCR_EL2.FWB == '1' then
            if s1desc.attrs != MemAttr_NC then
                result.hints = s1desc.hints;
            else
                result.hints = MemHint_RWA;
            end
        else
            result.hints = s1desc.hints;
        end
        result.transient = s1desc.transient;
    end
    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;
    end
    return result;
Library pseudocode for shared/translation/attrs/LongConvertAttrsHints

```c
// 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

```c
// 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

```java
// S1CacheDisabled()
// ===============

boolean S1CacheDisabled(AccType acctype)
if ELUsingAArch32(S1TranslationRegime()) then
    if PSTATE.EL == EL2 then
        enable = if acctype == AccType_IFETCH then HSCTL.R.I else HSCTL.R.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

```java
// 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>;
        elsif 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';
            memattrs.outershareable = SH == '10';
        else
            memattrs = MemoryAttributes UNKNOWN;    // Reserved
    return MemAttrDefaults(memattrs);
```

Library pseudocode for shared/translation/attrs/S2CacheDisabled

```java
// 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_EL2.I else SCTLR_EL2.C;
return disable == '1';
```
Library pseudocode for shared/translation/attrs/S2ConvertAttrsHints

// 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 !IsZero(attr);
MemAttrHints result;
if HCR_EL2.FWB=='0' && 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;
  endcase
result.transient = FALSE;
return result;

Library pseudocode for shared/translation/attrs/ShortConvertAttrsHints

// 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;
  endcase
result.transient = FALSE;
return result;
Library pseudocode for shared/translation/attrs/WalkAttrDecode

```c
// 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

```c
// 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

```c
// Have16bitVMID()
// ================
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/translation/translation/PAMax

```c
// 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;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
  return S1TranslationRegime(PSTATE.EL);

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 v31.04, AdvSIMD v29.02, pseudocode v2019-09_rc2_1, sve v2019-09_rc3 ; Build timestamp: 2019-09-27T18:00
Copyright © 2010-2019 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.